Spring 5-2004

FERRISBOT- An autonomous robot for IEEE competition

Brendan Powell MacDonald
University of Tennessee - Knoxville

Appendix E - UNIVERSITY HONORS PROGRAM
SENIOR PROJECT - APPROVAL

Name: Brendan MacDonald

College: Engineering Department: Electrical

Faculty Mentor: Dr. Laura Morris Edwards

PROJECT TITLE: FERRISBOT - An autonomous robot for IEEE competition

I have reviewed this completed senior honors thesis with this student and certify that it is a project commensurate with honors level undergraduate research in this field.

Signed: Laura Morris Edwards Faculty Mentor

Date: 5/3/04

General Assessment - please provide a short paragraph that highlights the most significant features of the project.

Comments (Optional):

Brendan participated on the ~~IEEE~~ Ferrisbot team that travelled to Greensboro NC for the annual robotics competitions. The rules were quite specific, & the team had to take into account several possible configurations plus environmental factors. He originally designed the line tracking system for the robot, but was ultimately involved in all components of a complex robotic system. Their efforts earned them third place out of 34 competitors throughout the Southeast.

Autonomous Robot Design

Brendan MacDonald
April 14, 2004

Faculty Advisors:
Dr. Laura Morris Edwards
Dr. Mongi Abidi

Abstract:

Developing both autonomous and remote controlled robots has become a major area of research recently with the purpose of protecting human life and instead sending hardware into the danger area. The hardware competition at The IEEE 2004 Southeast Conference concentrated on autonomous robot design. The robot needed to be able to start and stop at green and red lights, receive instructions through Morse code infrared transmission, make decisions about which path to follow, and pick up and drop off balls. These tasks are not directly applicable in everyday life, but provide the groundwork for more sophisticated robot design.
The entire team was involved in most tasks, but we each had specific responsibilities. My part of the project focused on providing the robot the ability to stay on the course. This involved building and testing hardware and software that enabled the robot to track the line. Also, I helped with the turn decisions the robot had to make based on the information it garnered from the infrared transmission.

IEEE ROBOTICS COMPETITION

Members: Will Curtis, Jonathon Britton, Brett Hatch, Scott Fields, Brian Nelson, Jacob DePriest, Justin Reed, Brendan MacDonald

Advisors: Dr. Mongi Abidi and Dr. Laura Morris Edwards

ECE 400
Senior Design

5/3/2004

Imaging, Robotics, and Intelligent Systems Laboratory
Dept. of Electrical and Computer Engineering
The University of Tennessee
Email: wcurtis1@utk.edu, sfields1@utk.edu, bnelson3@utk.edu, wreed1@utk.edu, bhatch@utk.edu, jbritton@utk.edu, jdepries@utk.edu, bmacdona@utk.edu

Contents

Table of Figures

1 INTRODUCTION

This report details the design and construction of a robot to compete in the IEEE Southeastern Conference Student Robotics Competition. This is an annual competition where robots are built by groups of IEEE student members to complete a predefined task that is laid out by the professional IEEE members holding the conference.

2 BACKGROUND

The task to be performed by the robot is to autonomously "hunt" small metal balls in a given order. Specifically, each of three animals (rabbit, duck, and deer) will be represented by a small metallic ball. These balls must be picked up in an order that is given through IR UART during the hunt. The balls must then be deposited in a box at the parking station. The total task can be broken up into three general stages: get hunting order, collect balls, and deposit balls.

2.1 HUNTING STAGES

2.1.1 Get Hunting Order

The robot will begin in the start box. When an LED cluster turns from red to green, the robot must drive through a 2'x 2' box to ensure it meets size requirements and proceed to the Morse code station shown in Figure 1 below. As the robot approaches the Morse code Station, another LED cluster will turn from green to red. When it turns red, the robot is to stop within 3 inches. At this time, the Morse code station will begin transmitting the hunting order using an 880nm Infrared LED to send data as a 300 bps, non-parity, UART transmission. The name of each animal will be spelled out in Morse code followed by a pause character. After all three animal names have been transmitted, a period is sent to signal the end of the sequence. The entire sequence is transmitted three times.

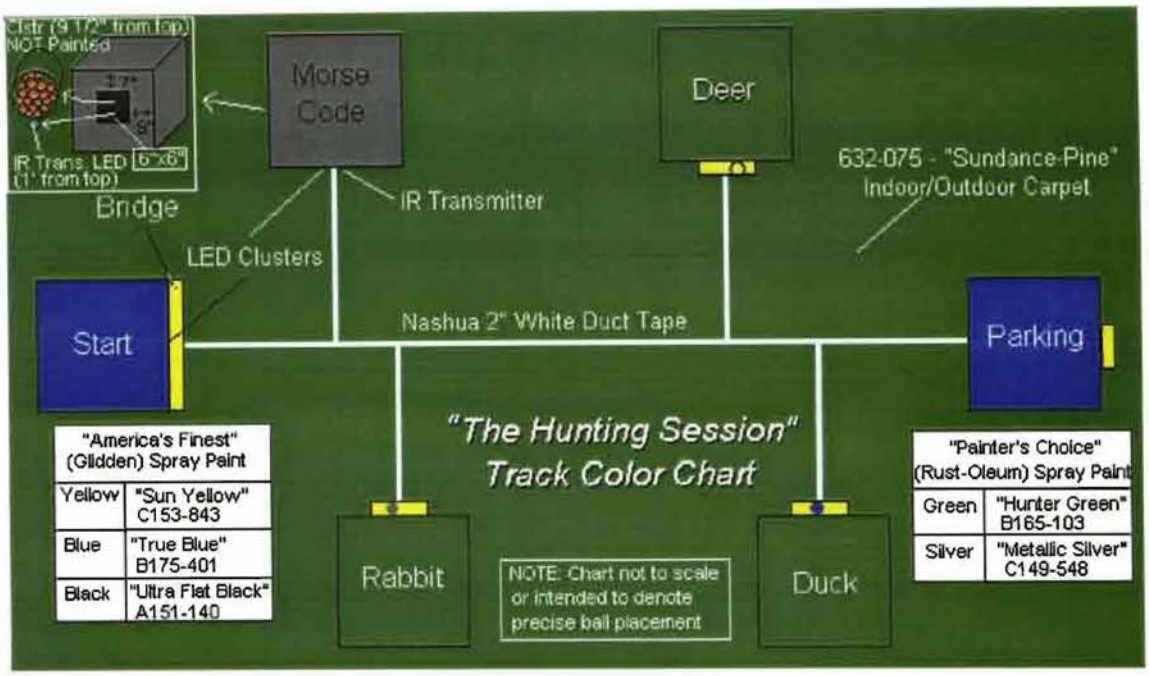

Figure 1: Track layout

2.1.2 Collect Balls

The next stage is the ball collection stage. In this stage, the hunting order that was determined in the last stage is, first, displayed on the LCD. The robot must then proceed to each station in the determined order and pick up a metal ball. All distances are unspecified so the only thing known about the stations is their relative positions.

2.1.3 Deposit Balls

The final stage is to deposit the balls in a container in the parking station. Again, since the hunting order and dimensions are unknown until run time, the final position is also unknown until runtime, so the robot must be smart enough to find its way to the parking station no matter where it is.

2.2 SCORING

Each run is allowed to take a maximum of 5 minutes. If it takes longer, the robot will be disqualified for that run. In addition, each run will be scored using a point system that adds or deducts points for the accuracy of the run. If the robot finishes in less than 5 minutes, an extra point is awarded for each second under this time limit. The list of point awards and deductions is below.

- Moving within 5 seconds of the traffic light turning green. (+20 pts)
- Moving completely out of the starting box (+10 pts)
- Display "STOP" at red light, "GO" at green light (+10 pts for each)
- Touching, or otherwise contacting the covered bridge (-5 pts)
- Moving after 5 seconds, but before 15 seconds of the traffic light turning green (+5 pts)

- Not moving within 15 seconds of the traffic light turning green (Disqualified for that run)
- Get to the Morse code station (+10 pts)
- Stopping within braking tolerance (3") when the traffic light turns red (+10 pts)
- List the name of each animal on the display unit, in correct order (+10 pts for each)
- Physical contact with the Morse code station (-5 pts)
- Get to each station (+10 pts for each)
- Capture each target in the correct order (+10 pts per target)
- Capture any target out of the correct order (0 pts)
- Display "DONE" after capturing the last target (+10 pts)
- Fitting completely in the parking area (+10 pts)
- Dispensing a ball into the cage (+15 pts per ball)
- Display "THE END" after dispensing all balls (+10 pts)
- Completing the individual run within 5 minutes (+10 pts)
- Remaining time, only if the robot completed the target sequence in the correct order (+1 point for each second)

The official rules can be seen at http://www.guilfordtechrobotics.org/.

3 ROBOT DESIGN AND CONSTRUCTION

A modular approach was taken to the design and construction of the robot. The following modules were designated:

- Microcontroller and Control Code
- Chassis and Drive
- Arm
- IR Morse Code Detector
- Red/Green Sensing
- Line Tracking
- Navigation and Positioning
- Aesthetics

In the following sections, the functionality of each module will be explained as well as its design and current status.

3.1 MICROCONTROLLER AND CONTROL CODE

The brain of the robot is the microcontroller that processes all of the sensor inputs and produces the necessary outputs to the display and motors. Ideally, the software that performs this function works reliably and is easy to modify throughout the design process.

3.1.1 Design Considerations

Much research preceded the selection of a microcontroller for the Ferrisbot. A number of boards were considered, including several using Basic Stamp, OOPic, and 68HC11 controllers and also one with a Field Programmable Gate Array (FPGA). After examining the number of inputs and outputs, the requirements for analog inputs, and the need for easy motor control, the Basic Stamp, OOPic, and FPGA boards were rejected, and the MIT Handyboard using the 68HC11 was chosen [Mar00].

There was a need in this project to monitor many sensors simultaneously, and the most natural way to do this seemed to be by using the built-in multitasking abilities of the Handyboard. By using a subsumption architecture, it was possible to implement multiple modules in a useful, efficient manner [Jon99].

Since the microcontroller needed to respond to inputs in real time, speed was often an issue. In addition to choosing an efficient control architecture, it was very important to eliminate floating point math and to limit all multiplication and division to powers-of-two. By implementing these optimizations, code that once ran sporadically became predictable and correct.

3.1.2 Implementation

Ferrisbot's microcontroller board is the Handyboard, a board designed by MIT specifically for robotics projects (see figure below). The Handyboard's processor is the Motorola HC6811, which runs at 2 MHz. The Handyboard also includes a number of useful integrated features: a 16x2 LCD display, built-in motor drivers, a serial programming interface, A/D converters, and a number of digital inputs and outputs [Mar00]. The Handyboard is supported by Interactive C, a language similar to ANSI C that allows fast programming, multi tasking, and easy interfacing with the Handyboard's capabilities [Kip03].

Figure 2: The MIT Handyboard shown without the Expansion Board

The top level of control code is the Ferrisbot state machine. The state machine keeps track of what sensor modules are needed at different points in the course. Each sensor module is designed to implement a behavior – it takes sensor readings for inputs and gives various

3.2.2 Implementation

The chassis consists of a twelve inch square piece of ABS plastic for a platform, two drive motors and wheels and a caster. The chassis was purchased pre-built and then modified to meet our specific needs. One major change made to the chassis was the replacement of the two DC motors. The two motors that came with the chassis did not meet our speed requirements, so they were replaced with different motors with a higher current rating. The platform sits six inches above the ground with the gearboxes suspended underneath and the drive motors are attached to the gearboxes. The gearboxes have a gear ratio of 146:1. The two twelve-volt motors are controlled via the microcontroller by a push-pull four channel motor driver called the L293D (http://www.alltronics.com/download/1330.pdf). With the 12 volt motors and 6-inch diameter wheels, this gives a speed of about two feet per second at a one hundred percent duty cycle.

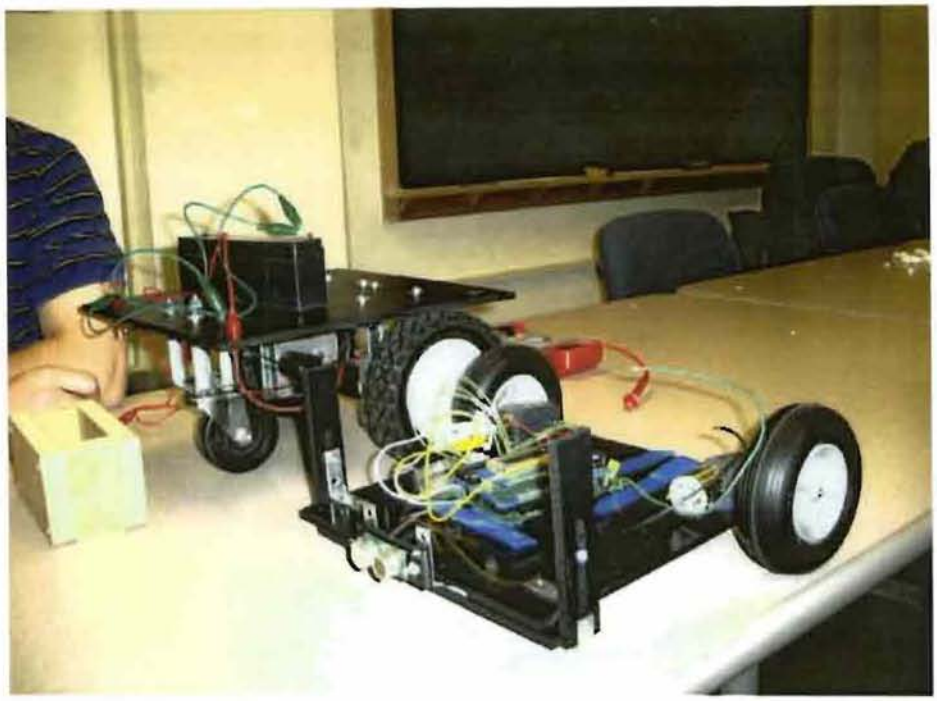

Figure 3: First chassis (front) shown side-by-side with its replacement (back)

The layout of the chassis is crucial, since all components must be supported by the chassis. Therefore, many things are taken into consideration when laying it out, including spacing and weight-distribution. Further, many components possess the ability to affect other parts of the robot, such as the magnetic components, for example.

The two drive motors are controlled by a set of intelligent H-bridge chips. These are on an external board that is mounted on the chassis. Power is routed from the main battery to the board and control signals are connected via two motor driver outputs from the Handyboard.

Table 2: Chassis and drive parts list

Part Description	Part Number

2 Channel Intelligent H-Bridge	Polydroids: MC1 Motion Controller Board Dual H-Bridge Driver

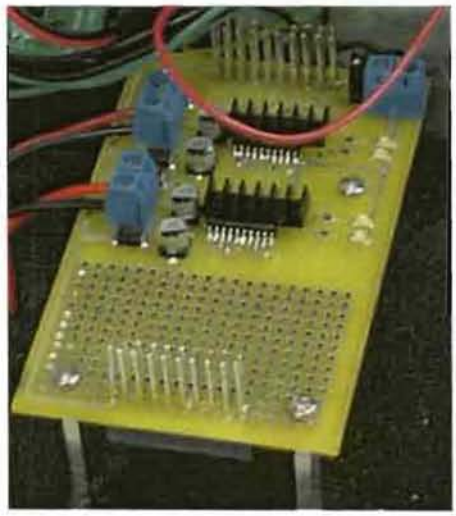

Figure 4: Motor driver board

3.2.3 Future Improvements

The chassis and drive worked as we expected. The robot was not limited by its speed or driving capability, but by the navigation system that controlled movement. The only possible improvement would be to consider a tank driven system that would reduce the turning radius.

3.3 ARM

The purpose of the arm is to pick up the one-inch diameter steel balls at each of the hunting stations. Part way through the design process for the arm the box specifications were changed such that the balls would sit at the top of the three-inch box and not in the bottom of the box. Thus the arm design became less "arm-like" and more "platform-like."

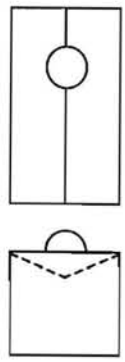

Figure 5: The balls to be retrieved sat partially recessed into the top of a hunting station box

3.3.1 Design Considerations

Multiple designs were considered to implement the task of collecting the balls. Various methods for picking up the balls were considered including: permanent magnets, electromagnets, vacuum, claws, and scoops. After considering reliability and practicality, the choice was narrowed down to permanent magnets and electromagnets. Due to the difficulty of separating the balls from permanent magnets, it was first thought that the best implementation would be with electromagnets. However, after attempts with both off-the-shelf and homemade versions, it was determined that the electromagnets of usable size were not powerful enough to reliably pick-up the balls. Thus it was decided that permanent magnets would be the best option.

Once the permanent magnets were chosen for the mechanism to pick up the balls, it was necessary to determine the best magnet size, magnet material, and method for getting the magnets over the box. Neodymium, alnico, and ferrite magnets in various shapes and sizes were all tested for strength and durability. It was found that the neodymium magnets were by far the strongest and those that were coated were reasonably durable. Additionally it was found that the round disc shaped magnets were best for picking up a single ball and long rectangular magnets were best for picking up multiple balls. Multiple designs were designed and tested. The first was a simple non-mobile platform. The advantage to this design is great speed (the robot only needs to drive over the box) and the reliability of the permanent magnets that are always "on." The basic platform design is shown in the figure below.

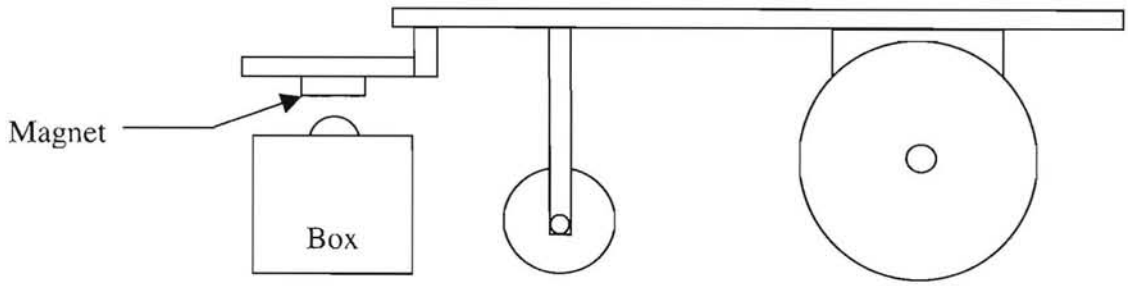

Figure 6: Platform arm design mounted on front of chassis

However after extensive testing it was determined that the simple platform was not reliable when the balls were all located in the same position in the box. This problem was further magnified when the robot had already picked up one ball and pulled over the next box. The new ball would be attracted to the ball that had already been collected and attach to it rather than be pulled up to the actual magnets. This sometimes caused the new ball to be "scraped off" by the box when the robot backed up away from the box. The magnets were strong enough to hold on to two balls but oftentimes failed when a third ball was added. Additionally, it was difficult to devise a plan to remove the balls at the parking station. Thus it was determined that the magnets needed to be able to move.

The first designs with "magnet motion" were based on magnets placed on a motor driven wheel. These designs were named the "Ferris" wheel, the paddlewheel, and the merry-go-round. The travel of the magnets for each of these designs is shown in the figure below.

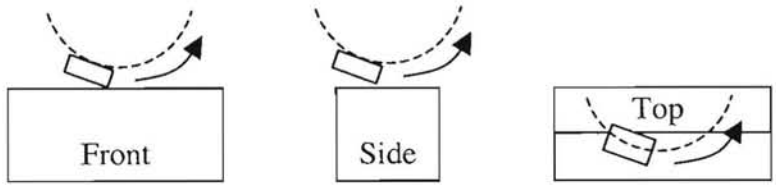

Figure 7: Various magnet motions were considered

After additional testing and thought, it was determined that each one of these designs was flawed in that it was possible for the ball to be in a location that might be missed by the magnets. In order for the design to be completely robust, the magnets must be able to sweep the entire box on a level plane rather than a circle. This led to the idea for the final design.

It was decided that the best option for sweeping the box would be to mount magnets on a moving belt. A timing belt and pulley system driven by a 12 VDC motor was selected for the system. Two additional pulleys were then added such that the belt would pass directly over the box and a fourth that would act to tension the belt. A rubber timing belt was chosen that was slightly larger than three times the box width. The magnets were then spaced evenly about the belt such that only one would pass over the box at a time. After the robot approached the box, the motor would turn for a specified amount of time and sweep the magnet across the width of the box. When the robot approached the parking station, the belt was rotated in the opposite direction and the balls were separated from the magnets with a metal shaft. The original design is shown in the figure below.

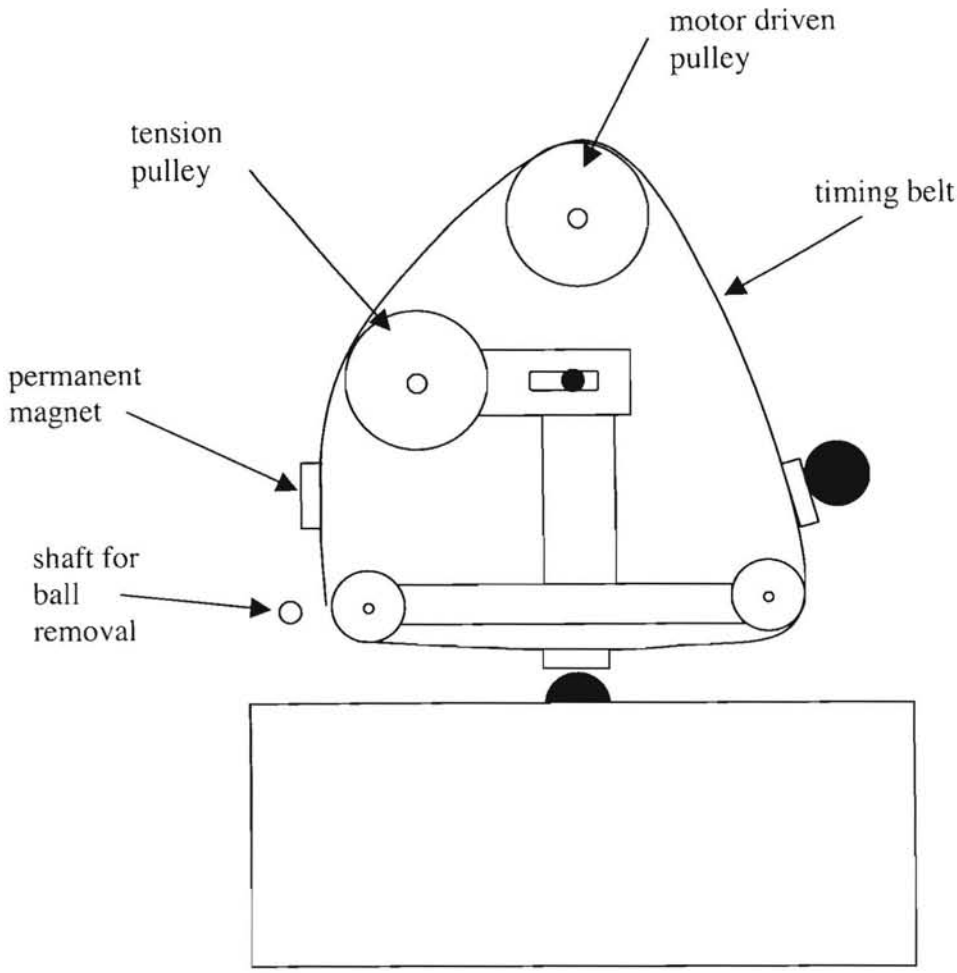

Figure 8: Belt design with original magnet positioning for retrieving balls

3.3.2 Implementation

While the design shown in the figure above was very robust and seemed to be fail proof, it was determined that the time it took to pick up and deposit the balls was too costly. The design used approximately 16 seconds of time while the robot was stopped to pick up and drop off all three balls. This was approximately 25% of the total run time goal of less than a minute.

The current design uses four magnets that are spaced much closer together. This is an attempt to get the speed benefits of the original platform while keeping the reliability of the belt system. The basis for the design is that two magnets are always on the lower portion of the belt. The robot drives over the box much like the platform design and then rotates the magnets one position while traveling to the next box. This saves the costly time spent at each box by removing the sweep as well as time to drop off the balls since the belt does not need to travel nearly as far. It also retains the reliability of the belt system by ensuring that each magnet will never encounter a third ball. This system cut the total time down to a mere 3.5

seconds, which is only 6% of the total run time! The final design is shown in the figures below.

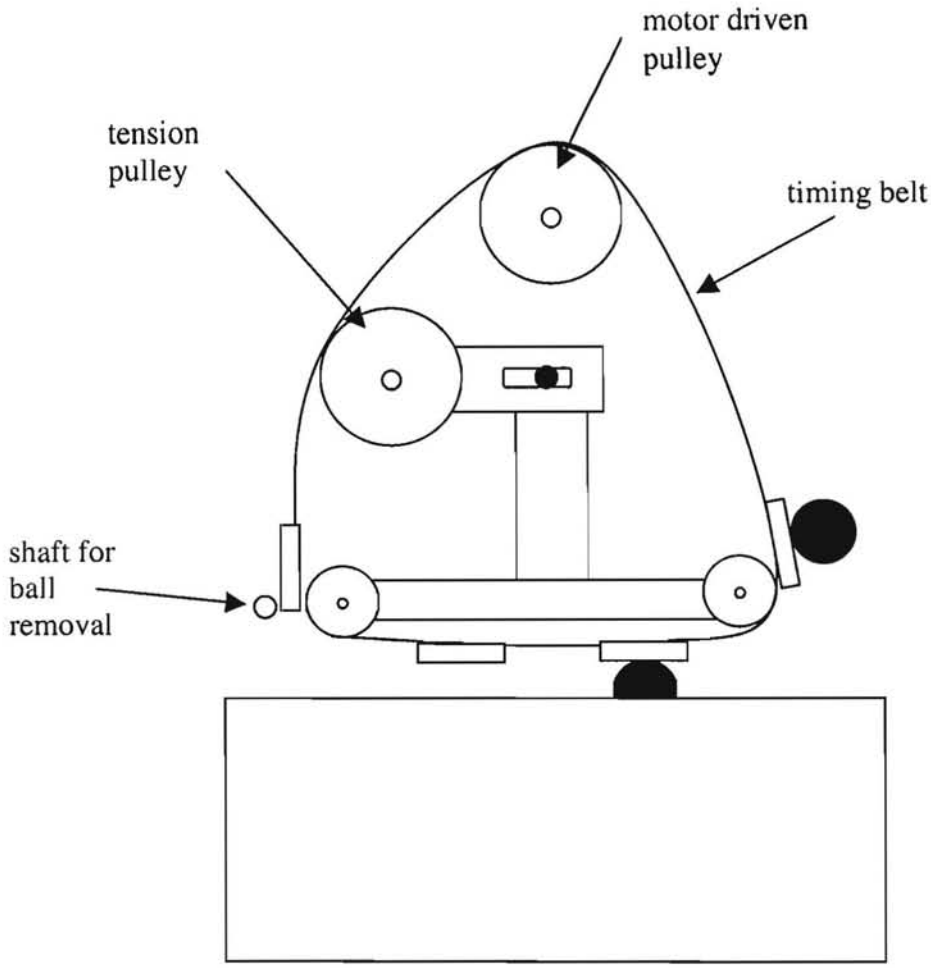

Figure 9: Belt design with updated magnet positioning for retrieving balls more quickly

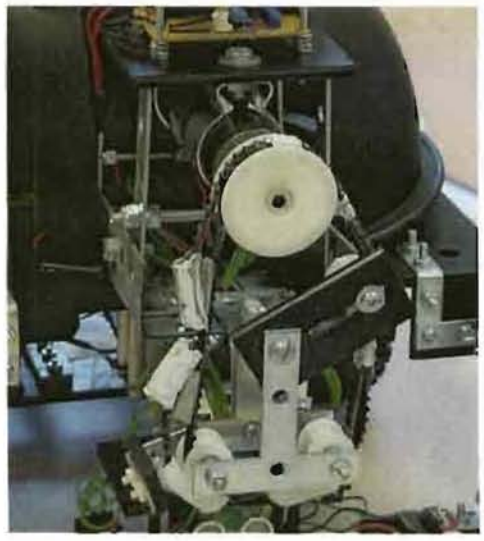

Figure 10: The final arm design shown mounted to the chassis

3.3.3 Future Improvements

Overall, the arm performance was nearly flawless. The major failures of the arm were due to processor issues that occurred when multiple processes were running simultaneously. In these instances such as when the when the robot was backing up and trying to turn the arm motor and belt would not rotate for the needed amount of time. Initially it had been decided to use time as the control for determining magnet location due to the belief that the resolution of the built in Hall Effect tachometer was unnecessary. However, in the future it might be a good idea to pursue using the tachometer. A second nuance that would be nice to change for future designs was the fact that the arm belt had to be manually aligned prior to each run. By adding more magnets with the same spacing around the entire belt, the setup time would be greatly reduced. Finally, due to the large amount of steel components involved in constructing the arm the weight was quite large. In the future it would be nice to reduce the weight

3.4 IR MORSE CODE DETECTOR

The purpose of the IR Module is to detect and decode the hunting order, and to pass this hunting order to the navigation code as a sequence of three numbers.

3.4.1 Design Consideration

The IR Module can be divided into two parts: hardware and software. The two main considerations when designing the hardware were to make it easy to integrate into the system and to make it adjustable to accommodate any ambient lighting conditions. The main consideration for the software was to make the code as small as possible so as not to waste any of the Handyboard's limited memory. Taking all of these factors into consideration, it was decided that using the MC68HC11's built in UART was the best option, as opposed to feeding the signal into a digital input and writing a timing loop to sample it and get the data. Using this UART, the code could be just a few lines long and the hardware could be plugged into an existing RJ11 jack on the Handyboard.

3.4.2 Implementation

The hardware contains an Infrared phototransistor with a pull-up resistor to give a voltage level. This resistor is a trim pot, which allows the adjustment of the sensitivity of the detector. That voltage level is fed to an op-amp along with a reference voltage to serve as a comparator. This reference voltage is generated from a voltage divider made out of a trim pot. This pot controls the threshold voltage of what is considered a one or zero. The output of the op-amp is then fed to a line driver to yield solid CMOS Voltage levels. A Schematic of the circuit and parts list can be seen below.
Note: The transistor in the schematic is a phototransistor, so the base will be supplied by an IR light source.

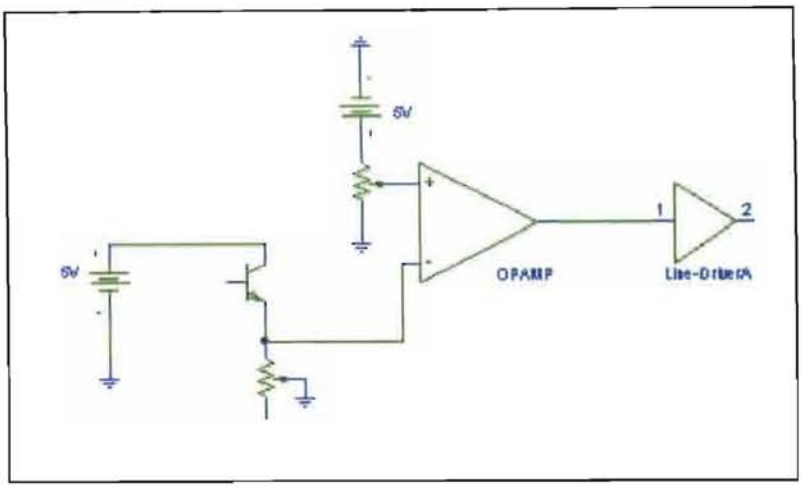

Figure 11: IR comparator schematic

The software is simply a function that sets the values in three control registers to set the UART to 300 bps, non-parity, 8 bits per character and puts the microcontroller into polling mode. Next a loop is used to read in the values being transmitted and then these values are compared to the known values to determine which of the six sequences is being sent. In the final version of the code, the known values did not correspond with the Morse code letters because of the way that the characters were being transmitted. The transmitter was using a timing loop to send the characters and did not insert the hold time after each character that the hardware UART required. As a result, the characters were not correct, when read in by the hardware UART. The same incorrect characters were read in each time, though, and the seventh character for each of the six sequences was unique so the sequence could be determined using only the seventh character. Once the sequence was determined, the correct sequence was set in a global integer array for use by the navigation code. The code for the IR module can be found in the Appendix.

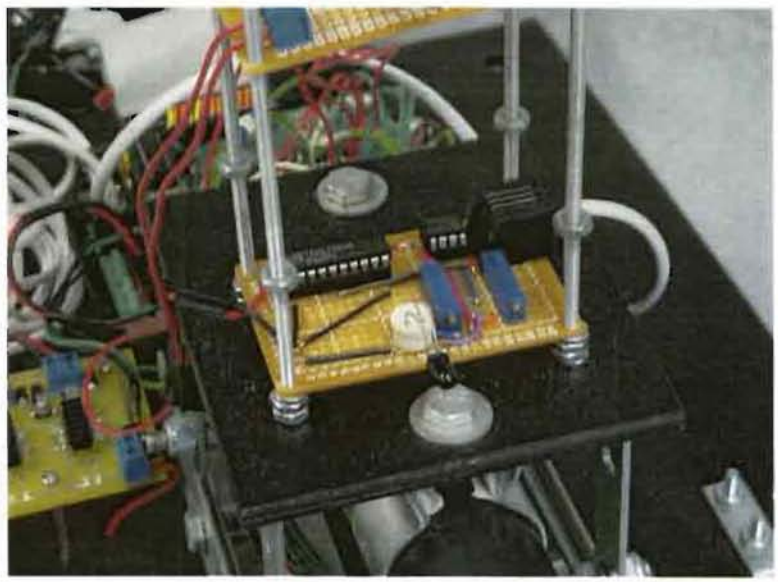

Figure 12: IR comparator circuit

Table 3: IR parts list

Part Description	Part Number
IR Phototransistor	DigiKey # QSD123QT-ND
Op-Amp	MC34072P
Line Driver	Phillips 74HCT541N

The hardware has been built on the breadboard and the code written. Currently, characters can be read in and printed out to the LCD, however, the data gets corrupted somewhere along the way and a few wrong characters are being printed out.

3.4.3 Future Improvements

One possible way to improve the IR module is to provide a method to read in the correct characters even if the hold times are not present between characters. This would involve feeding an output from the circuit into a digital input on the microcontroller board and having a timing loop to read in the bits manually. This would be very difficult to do using Interactive C because of the poor latency due to the fact that it is interpreted code. This non-deterministic flaw could possibly be solved by using compiled code, but this would greatly complicate the other code that depends on other features provided by Interactive C such as task switching. The advantage of having this feature would be the increased robustness of the design in that if the animal names were changed, the code could be changed accordingly without having to re-map each sequence manually to determine which characters were received by which transmission sequence.

3.5 RED/GREEN SENSING

Red/Green sensing was used to detect the state of the stop lights both at the stating station and at the Morse code station.

3.5.1 Design Considerations

There were several designs considered for the red/green visible light signal detection system. The three main concerns which helped determine the type of system chosen were:

1. Range
2. Ambient light rejection
3. Type of output

Range

The biggest issue when dealing with red/green sensing was the range of the detector. The specifications provided the current that would be flowing through the red and green lights, so the intensity of light that the red and green LEDs would be emitting could be determined. Further, the specifications for the competition track indicated that the robot needed to be able to distinguish between red and green light from 1 foot to 6 feet away from the IR station.

Ambient Light Rejection

Another concern in designing the sensing mechanism was that it must be able to reject a large percentage of the ambient light in the room. This was because of two reasons. Mainly, the sheer intensity of ambient light was easily more than that of the LED and would cause the electronics to saturate. In addition, the system could not be dependent on a certain ambient light level, since there was no way to know how bright the competition room would be.

Type of Output

Lastly, it was considered what type of output would be most efficiently implemented. For most of the duration of the project, the red/green sensor was built to give a digital (high/low) output. This was to be achieved by actually having two red/green sensing systems on each board. One would serve as a "DC" level or calibration device. It was not to be aimed at the area where it was expected that the actual light would be positioned. Thus, it essentially took an ambient light measurement, to which the other one would eventually be compared. Both the actual signal and ambient signal were fed to a comparator, which gave a 0V or 5V output to the controller board. One of the necessary modifications to the system was the addition of hysteresis at the input stage. This was controlled by an attenuating potentiometer, which greatly reduced noise in our digital signal.

Ultimately, the choice was made to feed an analog signal (directly from the frequency-to-voltage converter) to the controller board. This gave much more flexibility and diminished the effects of ambient and signal light variations, while also eliminating the possibility of an off-track entity (such as a red shirt, etc.) influencing the ambient sensor. This was nicely implemented in code by setting the trigger threshold to be different depending on the distance from the IR station. Therefore, the red/green code took the sonar distance reading into account.

The simplest designs were the first ones considered. Mainly, these consisted of either a photoresistor as a part of a resistive network or a photodiode with a bias network powered by the rails. The output would be a voltage probed from either side of the photoresistor or photodiode. Also, this required the use of a "gel" filter. Since red light gave the highest intensity and was at the edge of the visible light spectrum, it was the easiest to filter. Thus, using a red gel as a filter, it was possible to achieve some success with both of these setups. However, the usable range on these devices tended to stop at about 2 feet, which was not acceptable.

3.5.2 Implementation

A light-sensing array that was a light-to-frequency converter was located and used to resolve the aforementioned issues. This part was made by TAOS, inc., part TCS230 Optical Array. This part consisted of four photodiode arrays whose outputs were fed to a charge pump, which then created a varying-frequency 5Vpp square wave. A really nice feature of this part was that it had four built-in selectable filters. A multiplexer circuit selected either a red, green, blue, or clear array filter and sent the resulting output to the charge pump. Fortunately, this part had a usable range of over eight feet, which was even better than necessary.

To translate the output frequency into a usable voltage, LM2907 series frequency-to-voltage converters were used. These parts were made by National Semiconductor. Due to a small oversight, some initial issues existed with this part of the system while using the LM2907-8. The troubles arose from the fact that this version of the part has an internal ground, which requires the input wave to swing above AND below ground. After switching to the LM2907, the problem vanished. The final schematic is shown below.

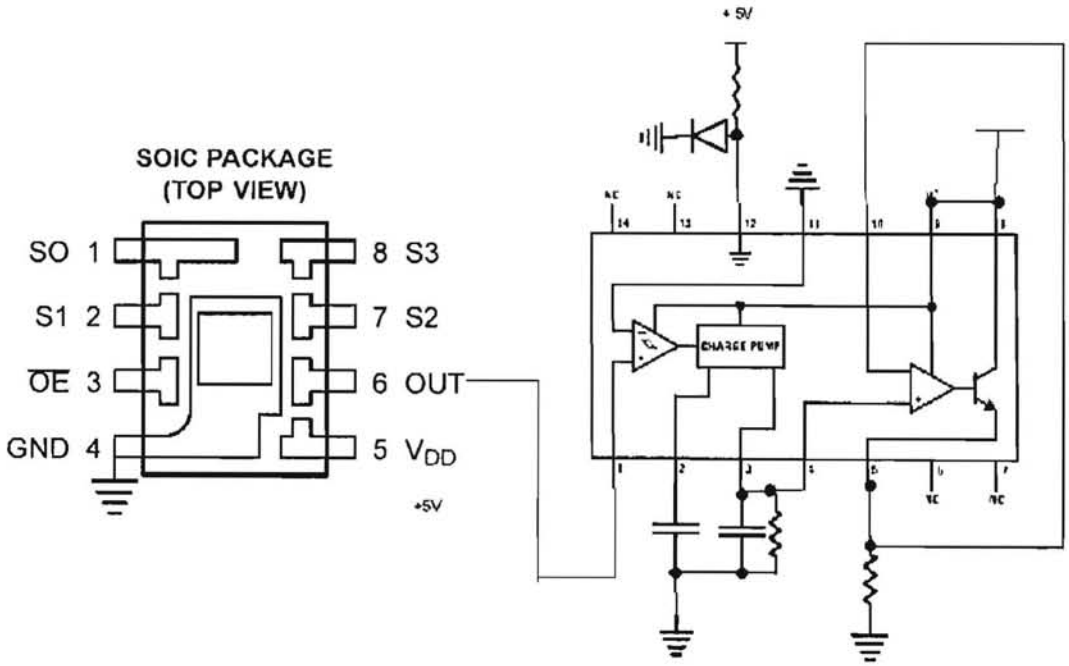

Figure 13: Red/green schematic

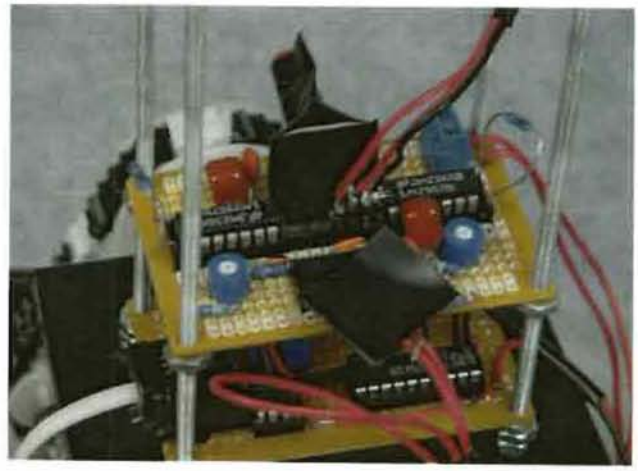

Figure 14: Red/Green sensing circuit

Table 4: Red/green sensing parts list

Part Description	Part Number

Light-to-Freq Converter	TSC230
Freq-to-Volt Converter	LM2907
Line Driver	74HCT541N
SurfBoard	Surface Mount 16-pin
Voltage Regulator	5V

3.5.3 Future Improvements

Since the entire red/green system worked without failure, the improvements needed are mainly for ease of use. For example, the sensor used at the starting station needed to be angled upward. During the competition, it was angled by bending the pins on the surface mount board and putting the top of the board in tension with a piece of tape so that it would remain at an angle. During the competition, however, it would work its way loose so that it had to be readjusted before each round. This caused an unnecessary waste of time, and would have been much easier if there had been a permanent solution to hold the angled board in place.

3.6 LINE TRACKING

While the rules did not specify the method of locating different stations, the most basic option was to track the white line. Some sort of camera could have been used with image processing, but the sensing could also effectively be carried out by a small number of pixel sensors.

3.6.1 Design Considerations

The line tracking module needed to give the robot the capability to traverse the course by staying visually aligned with the white line. Tracking is based on the principle that if a light source is positioned above the track, more light is reflected from the white tape than from the green carpet. To make use of this reflection, several visual light and infrared light emitters and sensors were evaluated. The first versions of the robot made use of visible light LED's and cadmium-sulfide (CDS) photoresistors.

The red LED's and photoresistors were positioned 1/8" above the track surface. Three of these emitter/receiver pairs were mounted on a low platform near the front of the robot and positioned such that one pair was in the middle of the tape while the other two were ¼" outside the edges. Each LED is positioned as close as possible to a photoresistor so that reflected light is incident on the sensor. Reflected light from the LED changes the resistance of the photoresistor, and the difference in reflection between the white tape and the green carpet is discerned in software. To keep non-reflected light from interfering with the sensing, both the LED's and the photo resistors are recessed in the lower platform (made of ¼" black ABS plastic). [Lea02]

The visual light emitters and sensors had many advantages over the infrared system. The difference in detected light between the tape and the carpet surfaces was greater. Also, it was easier to tell if the red LED's were functioning properly because, unlike the infrared emitters, it is possible to see their output. However, in order to have optimum performance from the visual light sensors the system had to be placed 1/8" above the ground. This low height

caused interference problems with the carpet that was specified for the competition. The infrared sensors performed best at ½" from the ground and they were used in the final version of the robot because of this great advantage.

3.6.2 Implementation

The line tracking system used at the competition consisted of infrared emitter and sensor pairs mounted ½" above the track surface. The schematics for connecting them to the microcontroller are shown below. [Lea02] The three front line tracking pairs were used to traverse the main line of the course as well as to approach the hunting station boxes after a turn off the main line was made. The two emitter/sensor pairs in the rear were used after a ball was retrieved from a hunting station for the robot to back up to the main white line. Their position relative to each other and the bottom of the robot is shown below in Figure 14.

It was important for the robot to maintain a rather straight course along the main white line so that the navigation sensors mounted further out would not detect the line being followed as a branch. Placement of the sensors is critical to ensure a straight path – moving the outer sensors too close to the tape makes the robot's movement choppy and slow, while moving the sensors too far results in large swings. Also critical was the speed of the two different motors driving the wheels when a correction in the path needed to be made – too quick a correction resulted in choppy movement and over-correction, while too slow a correction resulted in the robot veering off the line too far.

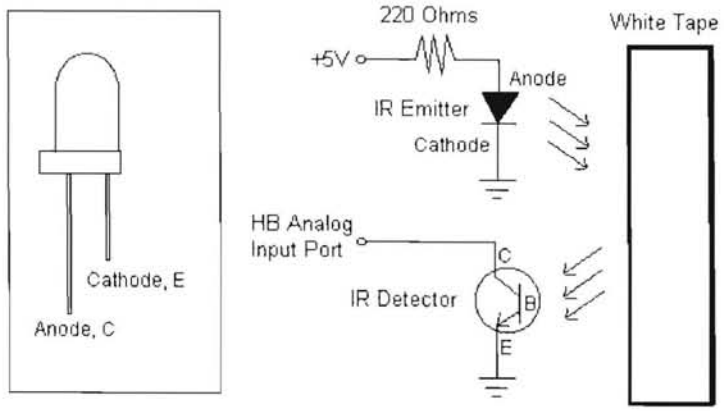

Figure 15: Schematic for connecting the IR emitter and sensors

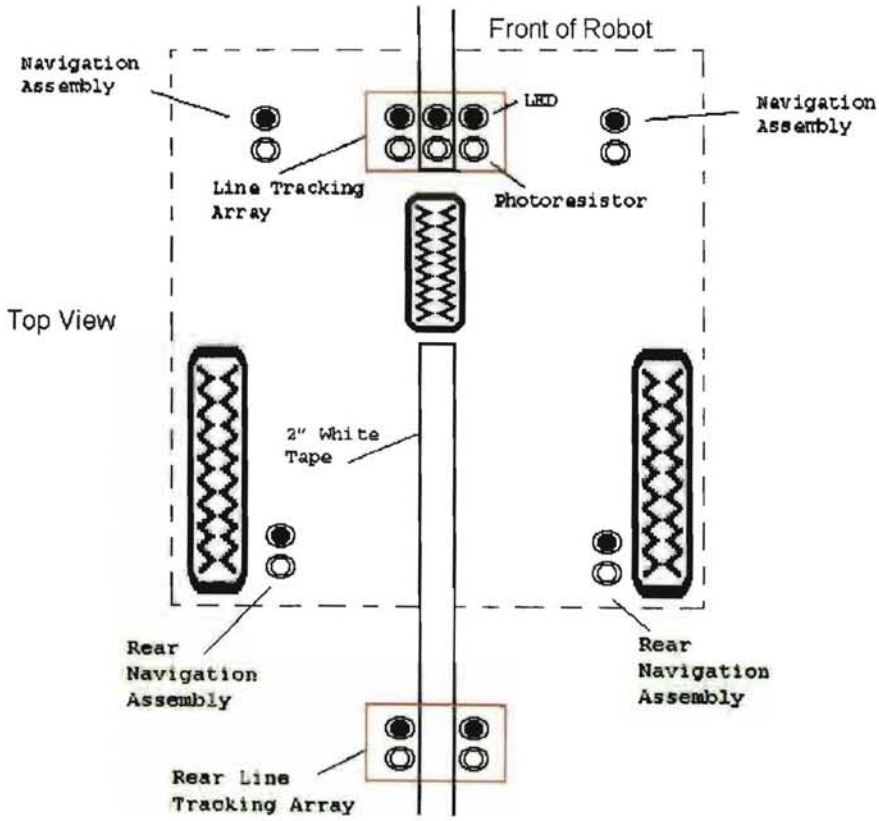

Figure 16: Line tracking LED array (and also navigation assemblies) is shown relative to the tape and chassis

Figure 17: Line tracking platforms mounted beneath the robot

Much testing was needed to determine the best combination of sensor placement and motor speed. In competition the outer sensors in the front and the back were placed ¼" from the outer edges of the line. The motor speed ratio was 3:2 when a correction in the path along the line needed to be made.

The system functioned correctly at the competition for at least one full run through the course. In testing it made many flawless runs through the course. When the robot lost the line in competition it was hard to tell if it was an error of the line tracking module or the navigation module because they were so closely intertwined. A bump in the carpet caused the robot to veer off the line and lose track of where it was so that it did not complete the course. It is likely the bump in the carpet raised the level of light detected by the robot from the carpet at this point so the software counted the bump as a branch off the main line.

Table 3: Line tracking parts list

Part Description	Part Number
High-Output IR LED	Radio Shack # 276-143
IR Phototransistor	Radio Shack # 276-145
220 Ohm Resistor	Radio Shack # 271-1313

3.6.3 Future Improvements

Improvements that could be made to the line tracking system would ensure that it could handle such problems as ripples in the carpet or extreme ambient light. It is not clear whether the best approach would be to add more of the same type of sensor pairs and have more complicated checks in the software, or if new hardware was needed.

Another option would be some sort of shielding for the visible light system so that it could be placed higher above the track because it seemed to give greater resolution between the carpet and tape surfaces. The improvements that could be made to line tracking are the same sort that are discussed in the navigation and positioning section. Chances are the error that caused the robot to lose the line occurred in the navigation module because of the line tracking module's ability to correct itself.

3.7 NAVIGATION AND POSITIONING

The navigation function is responsible for directing the robot between stations. The sensing considerations were the same as those in line tracking, and placement of the line sensors became the main design consideration.

3.7.1 Design Considerations

The first iteration of the turn sensing hardware added sensors to the front line tracking platform, placing another IR LED/phototransistor pair two inches to either side of the outer line tracking sensors. This implementation made smooth turns onto a branch by using a constant time delay between sensing the branch and executing the turn. One issue with such a design was that as the battery voltage changed, the speed of travel would change, and the

time delay would be incorrect to make smooth turns. Even a more important issue was that when traversing several legal track layouts, those in which the next turn to be taken was close to the just-exited branch, the hardware configuration would not be able to detect the branch and turn onto it.

To solve the close turn problem, a second set of navigation sensors was added on a rear platform mounted below the drive gearboxes. This allowed the entire set of legal track layout to be handled. To solve the time delay problem, several algorithms were tested, and it was eventually decided that the turn could happen as soon as the rear navigation sensors detected the branch. Details of this approach are discussed below in the Implementation section.

At the start of the design, it was clear that there would have to be some means of sensing stations remotely. IR ranging and sonar ranging sensors were considered for this task, and sonar was chosen to avoid any chance of IR interference with any of the light sensors on the robot. After testing the sonar ranger, it was determined that the unit was accurate down to about a millimeter, and with this level of precision the first approach was to use a single sonar sensor as the only sensor for lining up perfectly on a hunting station box. However, there were two problems with this design: first, the ranger had a minimum range of about 4 inches, and second the ball pickup device needed to be perfectly aligned in order to function properly. To solve both of these issues, bump sensors were integrated into the design.

3.7.2 Implementation

Navigation hardware is mounted on both the front and the rear of the robot. On the front, an additional IR LED and phototransistor are mounted several inches on either side of the line tracking hardware. On the rear, two IR LED/phototransistor pairs are mounted in a platform held in place beneath the drive motor gearboxes (see figure above). Inputs are digitized by the Handyboard A/D ports.

Navigation decisions are made by on a state machine that considers the origin, destination, and current position of the robot. As the robot progresses through the track, branches are detected and either noted for later reference or taken; any legal set of instructions and track layout will result in the robot making its way from the starting station to each of the hunting stations and then on to the parking station. Turns are divided up into two classes: close and non-close. Close turns occur when a branch that needs to be taken is so close to the previous branch that the front navigation sensors never see the branch to be taken. For this case, the turn is taken as soon as it is sensed by the rear navigation sensors. Non-close turns are the other case where the front navigation sensors will detect the branch to be taken. Since the branch will have been further away from the starting branch, the robot will have attained a higher rate of speed, and a turn based solely on the rear navigation sensors would result in an overshoot and poor alignment on the destination station. Thus, for the non-close case, the front navigation sensors are used to signal the robot to slow. The turn is then taken when the branch is sensed by the rear navigation sensors. Thus, the track can be traversed at high speeds, but all turns can be executed flawlessly. Special cases, as in 4 way intersections are handled correctly by the code by using a number of flags (see appendix for details).

The robot must approach and interact in a controlled manner with several stations in its environment, and to do so it uses a Devantech sonar ranger. When approaching the Morse

code station, the robot is not allowed to touch it. While the stoplight is supposed to signal the robot to stop, a failure in the stoplight circuitry could cause this signal to be missed and result in the robot coming into contact with the station, thus being disqualified. To prevent this, the sonar ranging unit is polled whenever the robot is on the Morse code branch, and the robot is stopped if it approaches too close to the station. When approaching the hunting stations and the parking station, the robot is preparing to pick up or deposit balls. This being the case, it needs to position itself precisely, and a slow speed is required. Thus, the sonar ranger is polled during the approach to any of these stations, and the robot is slowed (while still maintaining line tracking abilities) to facilitate a precise alignment (see figure below).

On the final approach to the hunting and parking stations, the sonar ranger has slowed the robot to a manageable speed, but the robot still needs to be certain of when contact has been made with the box. To this end, two bump sensors are mounted on the very front of the robot. When either of the bump sensors makes contact with a station, the robot shuts off its drive motors and can proceed with ball retrieval or deposit.

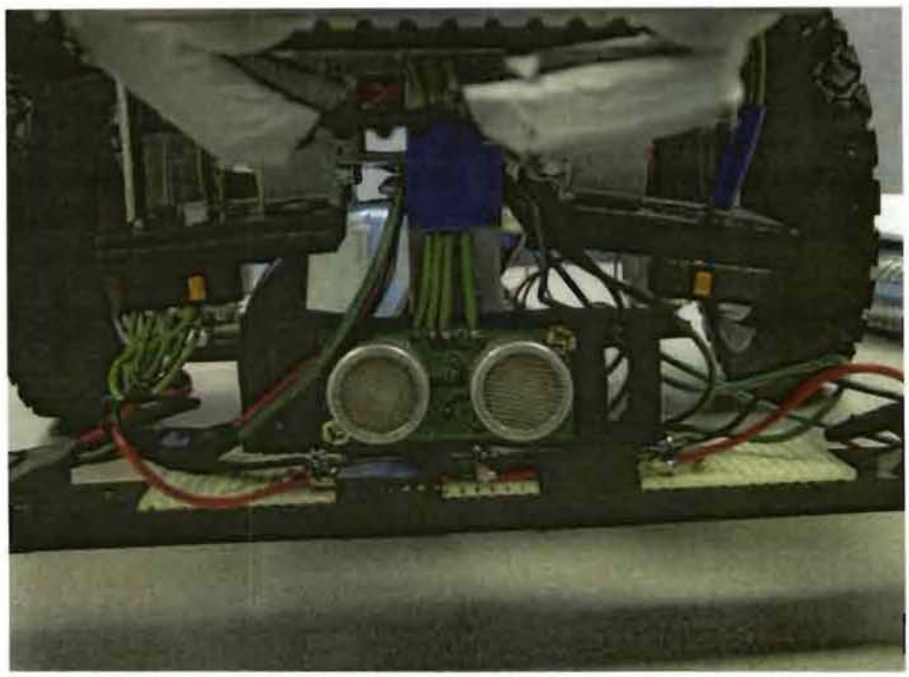

Figure 18: Sonar ranging sensor flanked by bump sensors

Table 5: Navigation and Positioning Parts list

Part Description	Part Number
High-Output IR LED	Radio Shack # 276-143
IR Phototransistor	Radio Shack # 276-145
220 Ohm Resistor	Radio Shack # 271-1313
Devantech SRF04 Ultrasonic Sonar Ranger	Acroname # R93-SRF04
Bump Switch Assembly Kit	Lynxmotion # BMP-01

3.7.3 Future Improvements

During the competition, it became clear that the turn sensors could deliver false readings from ripples in the track carpet. A more robust sensor design would have been able to function correctly even under these conditions. In the competition, these problems were resolvable by manually flattening the track before a run.

A future design would look at using visible light LEDs once again. It would be possible to correct the previous issues that visible light had by building an omni-directional skid and a light shield around the sensor. With such a structure, the sensors would be close to the track without becoming snagged on it, and light interference would also be a non-issue. Also, more IR LED/phototransistor combinations could be tested, as some of them may prove to be more resilient to wrinkles.

Overall, the positioning system worked very well, though the design could still be improved. The sonar sensor occasionally saw echoes from smooth surfaces, and this could lead to bad readings. Though these readings were compensated for and did not cause any trouble, a future design will look at eliminating this problem altogether. Using an IR ranger is one possible solution, though testing would be required to ensure that the emitted light did not interfere with other systems.

Another improvement would be to the bump sensors. Because they made direct contact with the boxes, and because they received a large number of these impacts throughout testing, they exhibited some reliability issues. During testing, one sensor was destroyed and had to be replaced. A better design would use more robust bump sensors or build some sort of cage around them to absorb the force of impacting the box.

3.8 AESTHETICS

With the main design laid out, attention was turned to the aesthetic design of the robot.

3.8.1 Design Considerations

There were several initial design considerations for the aesthetics of Ferrisbot. In the beginning the team was going to concede to leave the robot "as is" with only a simple fan mounted on the main chassis for cooling and some aesthetical effect. The next idea involved enclosing the Handy Board and some of the sensors in a box made out of the same material as the main chassis. Another consideration was to place a rounded object on top of Ferrisbot with a hinge to access the electronics when needed.

3.8.2 Implementation

The final design involves a piece of rounded plastic, much like half of a sphere, adapted from a normal serving bowl. The bowl is painted black and mounted upside-down on top of the robot, attached by a hinge to the rear part of the chassis for easy access to the Handy Board. This lid has a fan mounted on the top of it, forcing air into the enclosure. An LCD screen which displays current information about the robot is mounted on the back of the lid. It also has four buttons mounted on its left side: Handy Board power, fan power, stop button, and a start button. The main power button for the robot is located on its right side. Each button is wired to its respective switch on the robot and the wires are placed in such a way as not to

interfere with the opening of the lid. Two University of Tennessee power "T's" adorn the lid, one on each side, and a "University of Tennessee" sticker is on the rear part of the lid.

Figure 19: Chassis showing arm, line tracking, and navigation modules mounted

3.8.3 Future Improvements

There are a few improvements that could be made to the aesthetics of Ferrisbot. The first and most significant improvement to the robot would be to power the backlight of the LCD. This would not only make it easier to read, but it would also give it added visual flair. A purely cosmetic improvement would involve the addition of blue LEDs strategically placed inside and on the surface of Ferrisbot. A third and final aesthetic improvement would be the addition of a flexible tube to enclose the wires connecting the buttons on the lid to the electronics on the robot.

3.9 BUDGET

The total cost of materials used in the final design was $604. In the course of the design, however, a total of $2005 was spent on materials (this includes the $604 figure). This larger value includes all backup parts, test track materials, and experimental parts. A complete breakdown of parts by category is visible below in the bill of materials.

Table 6: Bill of Materials for the Ferrisbot design

Bill of Materials				
Category	**Qty.**	**Part Name/Number**	**Total Cost**	**Final design cost**
Aesthetics	1	Radio Shack Switches	$20	$20
Aesthetics	3	Stickers	$10	$10
Aesthetics	1	Salad Bowl	$3	$3

Arm	1	Misc. Lowe's/Radio Shack Arm Hardware	$100	$100
Arm	8	0.83" plastic pulley 57105K11	$57	$23
Arm	2	12VDC Globe gearhead motor 409A582	$40	$20
Arm	30	Various Magnets	$45	$12
Arm	1	Various Rubber and Urethane Belts	$21	$6
Arm	1	9.6V 1.8deg Stepper	$20	
Chassis/Drive	1	Max 97 Mobile Robot Base Only	$170	$170
Chassis/Drive	3	12V 2.2Ah battery	$30	$10
Chassis/Drive	1	Assorted inch angle brackets + bolts for chassis	$15	$10
Chassis/Drive	1	Sheet ABS for chassis	donated	
Chassis/Drive	2	MC1 Motion Controller Board Dual H-Bridge Driver	$70	$35
Chassis/Drive	20	L293D H Bridge	$60	
Chassis/Drive	3	Tamiya gearboxes	$72	
Chassis/Drive	2	Flange Mount Ball Transfer Systems 1" Ball NFMS	$30	
Chassis/Drive	6	Radio shack 12V and 9-18V DC motor (Brett)	$27	
Chassis/Drive	1	Pair 4.5" wheels	$23	
Chassis/Drive	2	3.25" RC aircraft wheels	$18	
Chassis/Drive	2	Narrow drive wheels	$5	
Chassis/Drive	2	12 VDC motors	$2	
Chassis/Drive	1	Misc gears	$1	
IR	10	IR LEDs QED123-ND	$5	$1
IR	2	IR logic discriminators QSE156-ND	$2	$1
IR	9	IR Photo Darlingtons	$15	
Line Track	15	IR Phototransistor/LED matched pair	$60	$27
Line Track	10	High Output IR LEDs 276-143	$18	
Line Track	15	IR Phototransistor QSD123QT-ND	$13	
Line Track	2	CDS photoresistor multi-pack	$3	
Microcontroller	1	Handyboard Backup GRHB-PC	$300	
Microcontroller	1	Handyboard - MC68H11 Microcontroller	$229	
Microcontroller	1	Expansion Board for Handy Board	$59	
Misc	1	Radio Shack Accessories	$60	$60
Misc	1	Parts from ECE Parts Store	$50	$25
Misc	1	Fan	donated	
Misc	1	12V 1A battery charger	$13	
Misc	1	Charger cord	$2	
Nav/Position	1	Devantech SRF04 Ultrasonic Range Finder	$33	$33
Nav/Position	2	Bumper Switch Assembly Kit BMP-01	$20	$10
Red/Green	10	RGB to freq photo diode array TCS230	$70	$14
Red/Green	8	Surfboard Alltronics 9163	$40	$10
Red/Green	12	IC Freq-to-voltage convertor Digikey LM2917N-8-ND	$23	$4
Red/Green	1	49mm Red 25 Lens B000050614	$11	
Red/Green	3	Red Photo Detector TSLB257R	$9	
Red/Green	3	Green Photo Detector TSLB257G	$9	
Red/Green	2	Green phototransistors PDV-V417-ND	$8	
Red/Green	2	IC quad comparator 296-12312-5-ND	$2	
Track/Testing	1	Wood, screws, carpet, paint for test track	$60	
Track/Testing	2	Red/green LED array	$40	
Track/Testing	10	Steel balls 9528K33	$12	
Total			**$2,005**	**$604**

4 REFERENCES

[Atm02] Atmel: *8-bit Microcontroller with 2K Bytes of In-System Programmable Flash.* <http://www.atmel.com/dyn/resources/prod_documents/DOC0839.PDF> 2002.

[Gui03] Guilford Tech Robotics: *Rules (Rev 11.26.03) Adobe PDF.* <http://www.guilfordtechrobotics.org/docs/download.php?filename=Rules11-26-03.pdf> November 2003.

[Kip03] KIPR: *Interactive C.* <http://www.kipr.org/ic/> November 2003.

[Jon99] Jones, Joseph L.; Flynn, Anita M.; Seigler, Bruce A.: *Mobile Robots: Inspiration to Implementation, 2nd Edition.* AK Peters Ltd., 1999.

[Lea02] K. Leang: *Two Inexpensive Line Tracking Sensors for the MIT Handy Board Microcontroller.* <http://www.leang.com/robotics/info/articles/linesen/index.html> March 2002.

[Mar00] Martin, Fred G.: *The Handyboard Technical Reference.* <http://www.handyboard.com/techdocs/hbmanual.pdf> November 2000.

[Mot04] Motorola: *M68HC11Microcontrollers: MC68HC11E Family Data Sheet.* <http://e-www.motorola.com/files/microcontrollers/doc/data_sheet/M68HC11E.pdf> 2004.

5 APPENDIX

5.1 TOP LEVEL CONTROL CODE

```
/* ferrisbot.ic
Description: Implements the states needed to complete the robot course.
        Controls the starting and stopping of sensor modules.
*/

//use with #ifdef and #endif. comment out DEBUG if not needed
//#define DEBUG

//local #defines - motors
#define LEFT_MOTOR 0
#define RIGHT_MOTOR 1
#define ARM_MOTOR 2

//local #defines - arm
#define RETRIEVE_TIME_A 1.5
```

```
#define RETRIEVE_TIME_B 1.
//2.6
#define DEPOSIT_TIME 6.0
//8.5

//local #defines - stop light
#define RED 0
#define GREEN 1

//globals - navigation
int IR_STATION = -1;
int DUCK = 0;
int DEER = 1;
int RABBIT = 2;
int PARKING_STATION = 3;

//local #defines - stations
#define STATION_A 0
#define STATION_B 1
#define STATION_C 2

//local #defines - sonar
#define SONAR_IR_DIST 250
#define SONAR_STATION_DIST 145

//drive motor control duty cycle (use this for cruising)
//NOTE: float division needs to be done by hand in line tracking!
int FULL_SPEED = 95;
int FULL_SPEED_BACK = 75;

//line/navigation sensing vars used globally
int left_tolerance, middle_tolerance, right_tolerance, nav_left_tolerance, nav_right_tolerance;
int back_left_tolerance, back_right_tolerance, back_nav_left_tolerance, back_nav_right_tolerance;

//track sensing ports - front and back
int nav_left_sensor=20,left_sensor=19, middle_sensor=18, right_sensor=17,nav_right_sensor = 16;
int back_nav_left_sensor=27,back_left_sensor=23, back_right_sensor=22, back_nav_right_sensor=21;
//A/D port 24 only works for high current :-/ 26, 4, 5, etc.

//red-green port (from 2 - 6 )
int red_green_sensor_1=2;
int red_green_sensor_2=3;

//globals for hunting stations
int station[3];

//globals for module control - note: #defines NOT global!
int line_track_motor_flag; //for line_track()
int line_track_left_motor;
int line_track_right_motor;
int navigate_motor_flag; //for navigate()
int navigate_left_motor;
int navigate_right_motor;
int last_station;
int sonar_stop_flag; //for sonar_stop()
int bump_stop_motor_flag;
int bump_stop_left_motor; //for bump_stop()
int bump_stop_right_motor;
int line_track_back_flag=0; // for deciding line tracking operation
int bump_stop_line_track_flag=1; //for using line_track while bump_module is going on
int red_green=0;//0 means less than 440 mm (1.5 feet) 1 means greater than

//modules and hardware used
#use "sonar_module.ic"
// sonar uses:
//  Digital Out 0
//  Digital In 7
#use "line_track.ic"
// line tracking uses:
// Analog In  16(L), 17(C), 18(R) (back uses 21(L), 22(R))
```

```
#use "bump_module.ic"
// bump uses:
// Digital In 8 (left bump)
// Digital In 9 (right bump)
#use "navigate.ic"
//navigate uses:
//Analog In 19(L), 20(R), (back uses 23(L), 24(R))
#use "IR.ic"
//IR uses serial port

//main() handles the basic linear functions and controls processes
void main()
{
   int color;

   //tolerances for competition track:
   nav_left_tolerance=19;
   left_tolerance=18;
   middle_tolerance=16;
   right_tolerance=13;
   nav_right_tolerance=18;
   back_nav_left_tolerance=18;
   back_left_tolerance=11;
   back_right_tolerance=13;
   back_nav_right_tolerance=11;

   //tolerances for competition track: window
   nav_left_tolerance=19;
   left_tolerance=15;
   middle_tolerance=12;
   right_tolerance=18;
   nav_right_tolerance=18;
   back_nav_left_tolerance=17;
   back_left_tolerance=10;
   back_right_tolerance=12;
   back_nav_right_tolerance=13;


   //for testing, calibrate is commented out and tolerances are hardcoded in
   calibrate();     //calibration routine for line_track
   printf("Press start button\n");

   while (1) {

      //wait on start button to begin autonomous run
      while(!(start_button()));

      //begin - parked and waiting on stop light
      printf("STOP!\n");

      //wait for green light
      check_stop_light(&color, red_green_sensor_1);
      while( color != GREEN ){
         check_stop_light(&color, red_green_sensor_1);
      }
      printf("GO!\n");

      //seek IR station and stop on stoplight. receive instructions
      seek_IR();

      recv_IR(); //real IR receive function
      //receive_IR();//fake IR receive function for testing

      //seek each hunting station and retrieve its ball
      seek( station[STATION_A] );
      sleep(1.);
      start_process(move_arm(RETRIEVE_TIME_A,100,0)); //multi task
      seek( station[STATION_B] );
      sleep(1.);
      start_process(move_arm(RETRIEVE_TIME_B,100,0)); //multi task
```

```
    seek( station[STATION_C] );
    sleep(1.);
    //no need to spin while going to drop off

    //done with hunting
    printf("DONE\n");

    //seek parking station and deposit balls
    seek( PARKING_STATION );
    move_arm(DEPOSIT_TIME,-100,1); //synchronous

    //end of run
    printf("THE END\n");
    sleep(5.0);
    irstation_flag=0, rabbit_flag=0, duck_flag=0, deer_flag=0;
    irstation_rabbit_flag=0, irstation_duck_flag=0, deer_rabbit_flag=0, deer_duck_flag=0;
  } //end while(1) that re-runs track
}

//seek_IR() seeks IR station and returns after stopping and displaying "STOP"
void seek_IR(){

  int pid0,pid1; //process IDs
  int color; //color of stop light

  //seeking IR station and stopping on red light/sonar too close
  #ifdef DEBUG
   sleep(2.);
  printf("Seeking: IR Station\n");
  #endif

  //run line tracking and navigate until navigate completes turn
  pid1 = start_process( navigate(IR_STATION) );
  pid0 = start_process( line_track() );
  defer();
  while(1) {

    //check for navigate() finish
    if( last_station == IR_STATION ){ break; }

    if( navigate_motor_flag ){
      motor(LEFT_MOTOR, navigate_left_motor);
      motor(RIGHT_MOTOR, navigate_right_motor);
    }
    else if( line_track_motor_flag ){
       motor(LEFT_MOTOR, line_track_left_motor);
       motor(RIGHT_MOTOR, line_track_right_motor);
     }
  }

  //kill navigate()
  kill_process(pid1);

  //run red/green, line tracking, sonar, to find IR station
  pid1 = start_process( sonar_stop(SONAR_IR_DIST) );
  defer();
  while(1) {
    check_stop_light(&color, red_green_sensor_2);
    //check for closer than 1.5ft and stop light
    if( !red_green && color == RED ) break;
    //emergency stop before hitting station
    if( sonar_stop_flag ) break;

    if( line_track_motor_flag ){
      motor(LEFT_MOTOR, line_track_left_motor);
      motor(RIGHT_MOTOR, line_track_right_motor);
    }
  }
  //motor reverse added to see if we can stop plowing over the IR station
  motor(LEFT_MOTOR,-30);
```

```
  motor(RIGHT_MOTOR,-30);
  //   sleep(0.05);
  ao();

  //kill line track and sonar
  kill_process(pid0);
  kill_process(pid1);

  printf("STOP!\n");

} //end seek_IR()

//test function for IR receiving
void receive_IR(){

  station[STATION_A] = RABBIT;
  station[STATION_B] = DEER;
  station[STATION_C] = DUCK;

  print_station( station[STATION_A] );
  print_station( station[STATION_B] );
  print_station( station[STATION_C] );
  printf("\n");

  sleep(2.0);

}// end receive_IR()

//seek(int station_name) seeks a hunting station and returns on contact with it.
// NOT for IR! ( use seek_IR() )
void seek(int station_name){

  int pid0,pid1;

  #ifdef DEBUG
    print_station( station_name );
  printf("\n");
  #endif

  //run line tracking and navigate until navigate gets back to main, then find new branch
  pid1 = start_process( navigate(station_name) );
  pid0 = start_process( line_track() );
  defer(); //give navigate() time to run so that immediately back up when motor() called below
  while(1) {

    //check for navigate() termination
    if( last_station == station_name ){
      ao(); //kill motors until they can be re-controlled
      break;
    }

    if( navigate_motor_flag ){
      motor(LEFT_MOTOR, navigate_left_motor);
      motor(RIGHT_MOTOR, navigate_right_motor);
    }
    else if( line_track_motor_flag ){
       motor(LEFT_MOTOR, line_track_left_motor);
       motor(RIGHT_MOTOR, line_track_right_motor);
     }
  }

  //kill navigate()
  kill_process(pid1);

  #ifdef DEBUG
    printf("switching on sonar for seek\n");
  #endif

  //initialize sonar to travel down branch and find box, then defer to bump sensors
  pid1 = start_process(sonar_stop(SONAR_STATION_DIST));
```

```
  defer(); //let sonar update before moving
  while(1) {
    if( sonar_stop_flag ){
      ao(); //kill motors until they can be controlled
      break;
    }
    else{
      if( line_track_motor_flag ){
        motor(LEFT_MOTOR,line_track_left_motor);
        motor(RIGHT_MOTOR,line_track_right_motor);
      }
    }
  }

  //kill line tracking and sonar
  //kill_process(pid0);
  kill_process(pid1);

  //use bump sensor to position on box
  pid1 = start_process(bump_stop());
  defer();
  while(1){
    if( bump_stop_motor_flag ){

      motor(LEFT_MOTOR, bump_stop_left_motor);
      motor(RIGHT_MOTOR, bump_stop_right_motor);
    }
    else{ //bump_stop_motor_flag == 0 (stop found)
      ao();
      break;
    }
  }

  //kill ze bump
  kill_process(pid1);

  //kill line tracking
  kill_process(pid0);

} //end seek()

//move_arm - moves arm for some time and duty cycle (special case for final==true)
void move_arm( float time, int duty, int final ){
  //use sleep only on pickups
  if( final ){
    sleep(0.5);
  }
  if( final ){
    motor(RIGHT_MOTOR,-50);
    sleep(0.1);
    ao();
  }

  motor(ARM_MOTOR, duty);
  sleep(time);
  off(ARM_MOTOR);

  if( final ){
    motor(LEFT_MOTOR,-50);
    sleep(0.2);
    ao();
  }
}

//print_station() decodes the station ID and prints text to display
void print_station( int mystation )
{
  if( mystation == DUCK )
    printf(" DUCK ");
  else if( mystation == DEER )
```

```
    printf(" DEER ");
  else if( mystation == RABBIT )
    printf(" RABBIT ");
  else if( mystation == PARKING_STATION )
    printf(" PARKING STATION ");
  else
    printf(" BAD STATION! ");
}

//check_stop_light returns the color of the stop light
void check_stop_light(int *color,int port){

  int val;

  val = analog(port);
  if(red_green==1 && val>125)
   *color = RED;
  if(red_green==1 && val<125)
   *color = GREEN;
  if(red_green==0 && val>135)
   *color = RED;
  if(red_green==0 && val<135)
   *color = GREEN;

}
```

5.2 BUMP MODULE CODE

```
/* bump_module.ic

Description: bump_stop() guides the motors slowly until
        both of the bump switches are touching something

        Hardware Used:
         Digital In 8 (left bump)
         Digital In 9 (right bump)

        Assumes the existence of globals:
        bump_stop_left_motor
        bump_stop_right_motor
        bump_stop_motor_flag //1 until both bump sensors touching, then 0

*/

void bump_stop(){

  int my_left_flag;
  int my_right_flag;

  bump_stop_motor_flag = 1;

  while(1){

    //check left
    if( !digital(8) ){ //if left not touching
      bump_stop_left_motor = line_track_left_motor>>1;
      my_left_flag = 0;
    }
    else{ //if left is touching
      bump_stop_left_motor = 0;
      //cause death of line track after first contact
      bump_stop_line_track_flag = 0;
      my_left_flag = 1;
    }

    //check right
    if( !digital(9) ){ //if right not touching
      bump_stop_right_motor = line_track_right_motor>>1;
      my_right_flag = 0;
    }
```

```
    else{ //if right touching
      bump_stop_right_motor = 0;
      my_right_flag = 1;
      //cause death of line track after first contact
      bump_stop_line_track_flag = 0;
    }

    //check both
    if( my_left_flag || my_right_flag ){
      bump_stop_motor_flag = 0;
      bump_stop_line_track_flag = 0;
    }

  } //end while(1)

} //end bump_stop()
```

5.3 IR MODULE CODE

```
/* ir.ic
Description: Interprets IR Morse Code instructions.
*/

void recv_IR(){
  int i;
  int temp;


  //disable pcode_serial
  poke(0x3c, 1);

  //Set baud rate to 300bps
  poke(0x102b, 0x35);

  //Set character size to 8 bits
  poke(0x102c, 0x10);

  //Select polling mode and enable reciever
  poke(0x102d, 0x04);

  //take reciever out of idle
  temp = peek(0x102e);
  temp = temp & 0x7f;
  poke(0x102e, temp);

  //poll status register to see when a character is ready
//  for(i=0;i<2;i++){
    for(i=0;i<7;i++){
      while(!(peek(0x102e) & 0x20));
      temp=peek(0x102f);
    }

    if(temp==0x7D){
      station[0]=0;
      station[1]=1;
      station[2]=2;
      printf("Duck, Deer, Rabbit\n");
    // break;
  }
    else if(temp==0xEC){
      station[0]=0;
      station[1]=2;
      station[2]=1;
      printf("Duck, Rabbit, Deer\n");
    // break;
  }
    else if(temp==0xA0){
      station[0]=1;
      station[1]=0;
      station[2]=2;
```

```
            printf("Deer, Duck, Rabbit\n");
          //  break;
        }
          else if(temp==0x8F){
              station[0]=1;
              station[1]=2;
              station[2]=0;
              printf("Deer, Rabbit, Duck\n");
            //  break;
          }
            else if(temp==0xF6){
                station[0]=2;
                station[1]=1;
                station[2]=0;
                printf("Rabbit, Deer, Duck\n");
              //  break;
            }
             else if(temp==0x06){
                  station[0]=2;
                  station[1]=0;
                  station[2]=1;
                  printf("Rabbit, Duck, Deer");
                //  break;
             }
              else {          //if it gets here, sequence was caught in middle of transmission and is incorrectly read in
                 // if(i==0)
                   // sleep(1.5);  //on 1st time through, if sequence not caught, wait 1.5 secs & try again
                 // else{
                     station[0]=2;  //on 2nd time through, if sequence missed, put in default and go
                     station[1]=1;
                     station[2]=0;
                     printf("Rabbit, Deer, Duck\n");}
              //  }
             //}
             //disable the receiver
    poke(0x102d, 0x00);

    //enable pcode_serial
    poke(0x3c, 0);
}
```

5.4 LINE TRACKING MODULECODE

```
/* line_track.ic

  tracks a white line on a darker surface continuously
  place on darker surface and hit stop
  place on tape and hit stop
  after tolerance is displayed hit stop

*/


//comment out DEBUG to remove print statements
//#define DEBUG

//turning correction values
// for FULL_SPEED 70: 70/1.4 = 50
// for FULL_SPEED 100: 75
#define TURN_SPEED_FORWARD 50
// for FULL_SPEED_BACK 50: 50/1.4 = 36
// for FULL_SPEED_BACK 75: 50
// for FULL_SPEED_BACK 100: 75
// it seems to work best when TURN_SPEED_BACK == 60 and FULL_SPEED_BACK == 75
#define TURN_SPEED_BACK 60

//#define SWEEP_TIME_MSECS 400L
//#define SWEEP 60

void line_track(){
```

```
int idle_count=0;//counter used to break out of stuck-at-go/go-backwards bugs

line_track_motor_flag=0;
/*   long start_sweep_time = mseconds();



 while (line_track_back_flag==1 &&
      analog(back_right_sensor)>back_right_tolerance &&
      analog(back_left_sensor) > back_left_tolerance){

    line_track_motor_flag=1;

    if( mseconds() < start_sweep_time + SWEEP_TIME_MSECS ){
       line_track_left_motor = SWEEP;
       line_track_right_motor = -SWEEP;
    }
    else{
       line_track_left_motor = -SWEEP;
       line_track_right_motor = SWEEP;
    }
  }
*/
  while(1){
    //increment idle counter
    idle_count++;

    //(backwards)find tape if bot skewed clockwise
    while(line_track_back_flag && analog(back_right_sensor)<back_right_tolerance){
       line_track_left_motor=-FULL_SPEED_BACK;
       line_track_right_motor=-TURN_SPEED_BACK;
       line_track_motor_flag=1;
       idle_count = 0;
       #ifdef DEBUG
         printf("turn left\n");
       #endif
    }
    //(backwards)cruise backwards
    while(line_track_back_flag &&
        (analog(back_left_sensor)>back_left_tolerance)&&
        (analog(back_right_sensor)>back_right_tolerance)&&
        (analog(middle_sensor)<middle_tolerance)) {
       line_track_left_motor=-FULL_SPEED_BACK;
       line_track_right_motor=-FULL_SPEED_BACK;
       line_track_motor_flag=1;
       idle_count = 0;
       #ifdef DEBUG
         printf("go straight\n");
       #endif
    }
    //(backwards)find tape if bot skewed counter clockwise
    while(line_track_back_flag && analog(back_left_sensor)<back_left_tolerance) {
       line_track_left_motor=-TURN_SPEED_BACK;
       line_track_right_motor=-FULL_SPEED_BACK;
       line_track_motor_flag=1;
       idle_count = 0;
       #ifdef DEBUG
         printf("turn right\n");
       #endif
    }

    //(backwards) manually break out of stuck-at-go-backwards cases
    if( line_track_back_flag && idle_count > 10 ){
       line_track_left_motor=-(FULL_SPEED_BACK);
       line_track_right_motor=-(FULL_SPEED_BACK);
       line_track_motor_flag=1;
       //don't set idle_count
    }

    //(forward)find tape if off to left
```

```
    while(!line_track_back_flag &&
        analog(right_sensor)<right_tolerance &&
        analog(nav_right_sensor)>nav_right_tolerance){
      line_track_left_motor=FULL_SPEED;
      line_track_right_motor=TURN_SPEED_FORWARD;
      line_track_motor_flag=1;
      idle_count = 0;
      #ifdef DEBUG
        printf("turn right\n");
      #endif
    }
    //(forward)cruise forward
    while(!line_track_back_flag &&
        analog(left_sensor)>left_tolerance &&
        analog(middle_sensor)<middle_tolerance &&
        analog(right_sensor)>right_tolerance){
      line_track_left_motor=FULL_SPEED;
      line_track_right_motor=FULL_SPEED;
      line_track_motor_flag=1;
      idle_count = 0;
      #ifdef DEBUG
        printf("go straight\n");
      #endif
    }
    //(forward)find tape if off to right
    while(!line_track_back_flag &&
        analog(left_sensor)<left_tolerance &&
        analog(nav_left_sensor)>nav_left_tolerance){
      line_track_left_motor=TURN_SPEED_FORWARD;
      line_track_right_motor=FULL_SPEED;
      line_track_motor_flag=1;
      idle_count = 0;
      #ifdef DEBUG
        printf("turn left\n");
      #endif
    }

    //(forward) manually break out of stuck-at-go cases
    if( !line_track_back_flag && idle_count > 10 ){
      line_track_left_motor=FULL_SPEED;
      line_track_right_motor=FULL_SPEED;
      line_track_motor_flag=1;
      //don't set idle_count
    }
  }
}

// Calibration subroutine to determine sensor tolerances
void calibrate(){
  //notape vars
  int nav_left_notape, left_notape, middle_notape, right_notape, nav_right_notape;
  int back_nav_left_notape, back_left_notape, back_right_notape, back_nav_right_notape;
  //tape vars
  int nav_left_tape, left_tape, middle_tape, right_tape, nav_right_tape;
  int back_nav_left_tape, back_left_tape, back_right_tape, back_nav_right_tape;

  while(!stop_button()){
    //calibrate no tape for front sensors
    nav_left_notape=analog(nav_left_sensor);
    left_notape=analog(left_sensor);
    middle_notape=analog(middle_sensor);
    right_notape=analog(right_sensor);
    nav_right_notape=analog(nav_right_sensor);
    //calibrate no tape for rear sensors
    back_nav_left_notape = analog(back_nav_left_sensor);
    back_left_notape = analog(back_left_sensor);
    back_right_notape = analog(back_right_sensor);
    back_nav_right_notape = analog(back_nav_right_sensor);

    printf("NoTp: %d %d %d %d %d:%d %d %d %d\n",
```

```
        nav_left_notape,
        left_notape,
        middle_notape,
        right_notape,
        nav_right_notape,
        back_nav_left_notape,
        back_left_notape,
        back_right_notape,
        back_nav_right_notape);
    sleep(.1);
}
tone(800.,1.);
while(!stop_button()){
    //calibrate tape for front sensors
    nav_left_tape=analog(nav_left_sensor);
    left_tape=analog(left_sensor);
    middle_tape=analog(middle_sensor);
    right_tape=analog(right_sensor);
    nav_right_tape=analog(nav_right_sensor);
    printf("f Tape: %d %d %d %d %d\n",
        nav_left_tape,
        left_tape,
        middle_tape,
        right_tape,
        nav_right_tape);
    sleep(.1);
}
tone(800.,1.);
while(!stop_button()){
    //calibrate tape for back nav sensors
    back_nav_left_tape=analog(back_nav_left_sensor);
    back_nav_right_tape=analog(back_nav_right_sensor);
    printf("b Tape: %d %d\n",
        back_nav_left_tape,
        back_nav_right_tape);
    sleep(.1);
}
tone(800.,1.);

while(!stop_button()){
    //calibrate tape for back sensors
    back_left_tape=analog(back_left_sensor);
    back_right_tape=analog(back_right_sensor);
    printf("b Tape: %d %d\n",
        back_left_tape,
        back_right_tape);
    sleep(.1);
}
tone(800.,1.);


//for white line tape value is less than notape value
//set tolerances for front sensors
nav_left_tolerance=nav_left_tape+(nav_left_notape-nav_left_tape)/2;
left_tolerance=left_tape+(left_notape-left_tape)/2;
middle_tolerance=middle_tape+(middle_notape-middle_tape)/2;
right_tolerance=right_tape+(right_notape-right_tape)/2;
nav_right_tolerance=nav_right_tape+(nav_right_notape-nav_right_tape)/2;
//set tolerances for back sensors
back_nav_left_tolerance=back_nav_left_tape+(back_nav_left_notape-back_nav_left_tape)/2;
back_left_tolerance=back_left_tape+(back_left_notape-back_left_tape)/2;
back_right_tolerance=back_right_tape+(back_right_notape-back_right_tape)/2;
back_nav_right_tolerance=back_nav_right_tape+(back_nav_right_notape-back_nav_right_tape)/2;
printf("Tol: %d %d %d %d %d:%d %d %d %d\n",
    nav_left_tolerance,
    left_tolerance,
    middle_tolerance,
    right_tolerance,
    nav_right_tolerance,
    back_nav_left_tolerance,
```

```
        back_left_tolerance,
        back_right_tolerance,
        back_nav_right_tolerance);
  //do not remove this beep, because it clears a buffer(?) and fixes motor jolts later
  while(!stop_button());
  tone(800.,1.);
}
```

5.5 NAVIGATION MODULE CODE

```
/* navigate.ic

 Uses:
  navigate_motor_flag
  navigate_left_motor
  navigate_right_motor

  navigate_left_sensor
  navigate_right_sensor
  middle_sensor

*/

//comment out DEBUG to remove print statements
//#define DEBUG

//flags that denote a station has been passed (and is now toward the start)
int irstation_flag=0, rabbit_flag=0, duck_flag=0, deer_flag=0;
//flags that denote that there is a station directly across from the current station
int irstation_rabbit_flag=0, irstation_duck_flag=0, deer_rabbit_flag=0, deer_duck_flag=0;
//flags that a close station (on opposite side) exists if the bot turns in the specified direction
int close_left_turn=0, close_right_turn=0;

//block out sensing of branches after detecting new branch
#define BRANCH_SENSE_BLOCK_MSECS 300L
//???do we need a separate sense_block for fast and slow travel?

//factor that divides line tracking signals when needed for close branch
#define LINE_TRACK_FACTOR 1
//factor to divide FULL_SPEED by while in fast (first) part of turn
#define FAST_TURN_FACTOR 1
//factor to divide FULL_SPEED by while in slow (last) part of turn
#define SLOW_TURN_FACTOR 2

//buffers and memory for sensor readings
int nav_left_reading, last_nav_left_reading, second_last_nav_left_reading; //front nav has second_last since care about rising edges
int nav_right_reading, last_nav_right_reading, second_last_nav_right_reading;
int back_nav_left_reading, last_back_nav_left_reading; //back nav has no second_last since only care about good highs
int back_nav_right_reading, last_back_nav_right_reading;//(SET ALL last_*** flags to 0 after using in preparation for next use!)
long branch_sense_time; //sense time of an opposing branch (opposite side from start location)
long across_sense_time; //sense time of an across branch (fudged for fun! :)
int turn_sensed; //flag signaling that turn has been sensed by front nav - now will wait for rear nav to see

void navigate(int station){
  navigate_motor_flag = 0;

  //preset motor flags for reverse line tracking (every option below must set it off when not needed)
  line_track_back_flag = 1;

  //reset remembered flags and times
  nav_left_reading = 0;
  last_nav_left_reading = 0;
  second_last_nav_left_reading = 0;

  nav_right_reading = 0;
  last_nav_right_reading = 0;
  second_last_nav_right_reading = 0;

  back_nav_left_reading = 0;
  last_back_nav_right_reading = 0;
```

```
  back_nav_right_reading = 0;
  last_back_nav_right_reading = 0;

  turn_sensed = 0;

  if (station == IR_STATION){  //IR_STATION arg called from start of course (no last_station)
    line_track_back_flag = 0;

    #ifdef DEBUG
     printf("IRSTATION\n");
    #endif

    travel_down_to_ir();
    turn_left();
    irstation_flag = 1;
    last_station = IR_STATION; //signal to ferrisbot

  } //end station == IR_STATION

  else if (station==DEER) {

//      #ifdef DEBUG
        printf("DEER\n");
//      #endif

      //line track backwards until hit line
      while( ! (analog(back_nav_left_sensor) < back_nav_left_tolerance &&
            analog(back_nav_right_sensor) < back_nav_right_tolerance) );

      line_track_back_flag=0;

      if (last_station == IR_STATION) {
        close_left_turn = 0;
        turn_right();
        travel_down_to_adjacent_deer();
        turn_left();
      }

      else if(last_station==DUCK ){
          if (deer_duck_flag){
            close_right_turn = 0;
            close_left_turn = 0;
            turn_right();
            turn_right();
          }
          else if( deer_flag ){ //backtrack
              duck_flag = 0;
              //no across flag to check
              close_left_turn = 0; //should have been null already
              turn_right();
              travel_up_to_opposing_deer();
              turn_right();
           }
           else{ // deer_flag == 0
              close_right_turn = 0;
              turn_left();
              travel_down_to_opposing_deer();
              turn_left();
           }
       }

       else{ //last_station == RABBIT)
          if( deer_rabbit_flag ){
            close_right_turn = 0;
            close_left_turn = 0;
            turn_right();
            turn_right();
          }
          else if( deer_flag ){ //backtrack
```

```
                rabbit_flag = 0;
                //no across flag to check
                close_left_turn = 0;
                turn_right();
                travel_up_to_opposing_deer();
                turn_right();
              }
              else{ //deer_flag == 0
                close_right_turn = 0;
                turn_left();
                travel_down_to_opposing_deer();
                turn_left();
              }
            }// end last_station == RABBIT

            deer_flag = 1;
            //signal for ferrisbot that bot is approaching DEER
            last_station=DEER;

        } // end station == DEER

        else if (station == DUCK){    //DUCK arg called from some station
//          #ifdef DEBUG
            printf("DUCK\n");
//          #endif

            //line track backwards until hit line
            while (! (analog(back_nav_left_sensor) < back_nav_left_tolerance &&
                  analog(back_nav_right_sensor) < back_nav_right_tolerance ));

            line_track_back_flag=0;

            //decide on turns and execute
            if (last_station==IR_STATION){
              if( irstation_duck_flag ){ //duck station directly across
                close_right_turn = 0;
                close_left_turn = 0;
                turn_right();
                turn_right();
              }
              else if( duck_flag ){ //already passed duck station
                irstation_flag = 0;
                //no across flag to check
                close_right_turn = 0; //forget unused close turns
                turn_left();
                travel_up_to_opposing_duck();
                turn_left();
              }
              else{ //duck_flag == 0
                close_left_turn = 0; //forget unused close turns
                turn_right();
                travel_down_to_opposing_duck();
                turn_right();
              }
            } //end last_station == IR_STATION

            else if(last_station == DEER){
              if (deer_duck_flag){
                close_right_turn = 0;
                close_left_turn = 0;
                turn_right();
                turn_right();
              }
              else if( duck_flag ){
                deer_flag = 0;
                //no across flag to check
                close_right_turn = 0; //should be null already
                turn_left();
                travel_up_to_opposing_duck();
                turn_left();
```

```
            }
            else{ //duck_flag == 0
              close_left_turn = 0;
              turn_right();
              travel_down_to_opposing_duck();
              turn_right();
            }
          }//end last_station == DEER
          else{ // last_station == RABBIT
            close_right_turn = 0;
            turn_left();
            travel_down_to_adjacent_duck();
            turn_right();
          } //end last_station == RABBIT

          duck_flag = 1;
          last_station=DUCK; //signal to ferrisbot

        } //end station == DUCK

        else if(station==RABBIT){
//        #ifdef DEBUG
          printf("RABBIT\n");
//        #endif

          //line track backwards until hit line
          while (! (analog(back_nav_left_sensor) < back_nav_left_tolerance &&
                analog(back_nav_right_sensor) < back_nav_right_tolerance ));

          line_track_back_flag=0;

          if (last_station==IR_STATION){
            if( irstation_rabbit_flag ){
              close_right_turn = 0;
              close_left_turn = 0;
              turn_right();
              turn_right();
            }
            else if( rabbit_flag ){
                irstation_flag = 0;
                //no across flag to check
                close_right_turn = 0;
                turn_left();
                travel_up_to_opposing_rabbit();
                turn_left();
              }
              else{ //rabbit_flag == 0
                close_left_turn = 0;
                turn_right();
                travel_down_to_opposing_rabbit();
                turn_right();
              }
          }//end last_station==IR_STATION

          else if (last_station==DUCK){
              duck_flag = 0;
              if( deer_duck_flag ){
                deer_flag = 0;
              }
              else if( irstation_duck_flag ){
                  irstation_flag = 0;
                }
               close_left_turn = 0;
              turn_right();
              travel_up_to_adjacent_rabbit();
              turn_left();
            }//end last_station == DUCK
            else{ //last_station==DEER
              if( deer_rabbit_flag ){
                close_left_turn = 0;
```

```
                close_right_turn = 0;
                turn_right();
                turn_right();
              }
              else if( rabbit_flag ){
                deer_flag = 0;
                if( deer_duck_flag ){
                  duck_flag = 0;
                }
                close_right_turn = 0;
                turn_left();
                travel_up_to_opposing_rabbit();
                turn_left();
              }
              else{ //rabbit_flag == 0
                close_left_turn = 0;
                //no across flags to check
                turn_right();
                travel_down_to_opposing_rabbit();
                turn_right();
              }
            }//end last_station == DEER

            rabbit_flag = 1;
            last_station=RABBIT; //signal to ferrisbot

          } // end station == RABBIT

          else{// station==PARKING_STATION
            #ifdef DEBUG
              printf("entering navigate parking\n");
            #endif

            //line track backwards until hit line
            while (!((analog(back_nav_left_sensor) < back_nav_left_tolerance) &&
                  (analog(back_nav_right_sensor) < back_nav_right_tolerance) ));
            line_track_back_flag=0;

            if ( last_station==DEER){ //(won't be IR station)
              turn_right();
            }
            else{// if (last_station==RABBIT || last_station==DUCK)
              turn_left();
            }

            //no need to set flags for parking_station

            //signal for ferrisbot that bot is approaching parking station
            last_station = PARKING_STATION;
          }
}

//turns from main line to left branch
void turn_left(){
  //  #ifdef DEBUG
  //    printf("nav turn left\n");
  //  #endif
  navigate_motor_flag=1;

  //stupidly move for some amount of time
  navigate_right_motor = FULL_SPEED>>FAST_TURN_FACTOR;
  navigate_left_motor  = -(FULL_SPEED>>FAST_TURN_FACTOR);
  sleep(0.2);

  //move left while left nav sensor is off line
  //note strangeness: use of greater than sign to assign readings.
  while( analog(nav_left_sensor) > nav_left_tolerance ){
    navigate_right_motor = FULL_SPEED>>FAST_TURN_FACTOR;
    navigate_left_motor  = -(FULL_SPEED>>FAST_TURN_FACTOR);
  }
```

```
  //continue moving right until middle sensor is on line
  while( analog(middle_sensor) > middle_tolerance ){
    navigate_right_motor = FULL_SPEED>>SLOW_TURN_FACTOR;
    navigate_left_motor = -(FULL_SPEED>>SLOW_TURN_FACTOR);
  }

  navigate_right_motor = 0;
  navigate_left_motor  = 0;

  //return control to line_track()
  navigate_motor_flag=0;
}

//turns from main line to right branch
void turn_right(){
  //   #ifdef DEBUG
  //     printf("nav turn right\n");
  //   #endif

  //take control of motors
  navigate_motor_flag=1;

  //stupidly move for some amount of time
  navigate_right_motor = -(FULL_SPEED>>FAST_TURN_FACTOR);
  navigate_left_motor  = FULL_SPEED>>FAST_TURN_FACTOR;
  sleep(0.2);

  //move right while right nav sensor is off line
  //note unusual use of greater than in setting the reading - leads to '1' for last_ vars
  while( analog(nav_right_sensor) > nav_right_tolerance ){
    navigate_right_motor = -(FULL_SPEED>>FAST_TURN_FACTOR);
    navigate_left_motor  = FULL_SPEED>>FAST_TURN_FACTOR;
  }
  //continue moving right until middle sensor is on line
  while( analog(middle_sensor) > middle_tolerance ){
    navigate_right_motor = -(FULL_SPEED>>SLOW_TURN_FACTOR);
    navigate_left_motor = FULL_SPEED>>SLOW_TURN_FACTOR;
  }

  navigate_right_motor = 0;
  navigate_left_motor = 0;

  //return control to line_track()
  navigate_motor_flag=0;
}

//traverse main line to IR
//called for start->IR
void travel_down_to_ir(){
  //watch for left turn
  while( 1 ){
    nav_left_reading = analog(nav_left_sensor) < nav_left_tolerance;
    nav_right_reading = analog(nav_right_sensor) < nav_right_tolerance;

    // set across flag if left and right overlap (new branch will have already been detected)
    if( nav_left_reading && last_nav_left_reading &&
      nav_right_reading && last_nav_right_reading ){
      if( rabbit_flag == 0 ){
        irstation_rabbit_flag = 1;
        rabbit_flag = 1;
        across_sense_time = mseconds();
        #ifdef DEBUG
          printf("set irstation_RABBIT_flag\n");
        #endif
      }
      else{ //rabbit_flag == 1
        if( mseconds() > across_sense_time + BRANCH_SENSE_BLOCK_MSECS ){
          irstation_duck_flag = 1;
          duck_flag = 1;
```

```
            #ifdef DEBUG
              printf("set irstation_DUCK_flag\n");
            #endif
          }
        }
      }

      //mark when turn first seen and go to slow speed
      if( turn_sensed==0 && nav_left_reading && last_nav_left_reading){
        turn_sensed = 1;
        navigate_left_motor = line_track_left_motor>>LINE_TRACK_FACTOR;
        navigate_right_motor = line_track_right_motor>>LINE_TRACK_FACTOR;
        navigate_motor_flag = 1; //will be set low by turn function
      }

      //watch for new branches entire time
      if ( nav_right_reading && last_nav_right_reading && second_last_nav_right_reading==0){
        //debounce branch sense
        if( mseconds() > branch_sense_time + BRANCH_SENSE_BLOCK_MSECS){
          branch_sense_time = mseconds();
          //set station flags only if detection is before turn sense
          if( turn_sensed == 0 ){
            if( rabbit_flag == 0 ){
              rabbit_flag = 1;
              #ifdef DEBUG
                printf("found the RABBIT\n");
              #endif
            }
            else{ //rabbit_flag==1
              duck_flag = 1;
              #ifdef DEBUG
                printf("found the DUCK\n");
              #endif
            }
          }
          //if branch detected after turn sense, set close right flag. nullified later if across
          else{ //turn_sensed
            close_right_turn = 1;
          }
        }
      }
      second_last_nav_right_reading = last_nav_right_reading;

      if( turn_sensed ){ //stuff for after turn sensed
        back_nav_right_reading = analog( back_nav_right_sensor) < back_nav_right_tolerance;
        back_nav_left_reading = analog( back_nav_left_sensor) < back_nav_left_tolerance;

        //watch for close rear branches after turn detected
        if( back_nav_right_reading && last_back_nav_right_reading ){
          close_left_turn = 1;
        }

        //exit loop for turn when back nav detects turn
        if( back_nav_left_reading && last_back_nav_left_reading ){
          break;
        }
        last_back_nav_right_reading = back_nav_right_reading;
        last_back_nav_left_reading = back_nav_left_reading;
      }
      last_nav_right_reading = nav_right_reading;
      last_nav_left_reading = nav_left_reading;
    } //end main while

    //clean up sense memory
    last_nav_left_reading = 0;
    last_nav_right_reading = 0;

    second_last_nav_left_reading = 0;
    second_last_nav_right_reading = 0;
```

```
  last_back_nav_left_reading = 0;
  last_back_nav_right_reading = 0;
}

//traverse main line to duck
//called for IR->DUCK and DEER->DUCK
void travel_down_to_opposing_duck(){
  if( close_right_turn && rabbit_flag ){ //use close turn
    //close_right_turn is maintained
    while( 1 ){
      nav_left_reading = analog( nav_left_sensor ) < nav_left_tolerance;
      back_nav_right_reading = analog( back_nav_right_sensor ) < back_nav_right_tolerance;

      //use line tracking to stay centered
      navigate_left_motor = line_track_left_motor>>LINE_TRACK_FACTOR;
      navigate_right_motor = line_track_right_motor>>LINE_TRACK_FACTOR;
      navigate_motor_flag = 1;

      //watch for a close branch
      if( nav_left_reading && last_nav_left_reading ){
        close_left_turn = 1;
      }

      //take turn immediately after sensing
      if( back_nav_right_reading && last_back_nav_right_reading ){
        break;
      }

      last_nav_left_reading = nav_left_reading;
      last_back_nav_left_reading = back_nav_left_reading;
    }//end while(1)
    navigate_motor_flag = 0;
  }
  else{ //use no close turn
    //make sure that if there was a close turn not taken, the flag is set appropriately
    if( close_right_turn ){
      rabbit_flag = 1;
      #ifdef DEBUG
        printf("found the RABBIT\n");
      #endif
      close_right_turn = 0;
    }

    //look for desired turn after other turn passed
    if( rabbit_flag ){
      branch_sense_time = mseconds() - BRANCH_SENSE_BLOCK_MSECS;
    }
    else{
      branch_sense_time = 0L;
    }
    while(1){
      nav_left_reading = analog(nav_left_sensor) < nav_left_tolerance;
      nav_right_reading = analog(nav_right_sensor) < nav_right_tolerance;

      //mark when turn first seen
      if( nav_right_reading &&
        last_nav_right_reading &&
        branch_sense_time > 0L &&
        (mseconds() > branch_sense_time + BRANCH_SENSE_BLOCK_MSECS) ){
        turn_sensed = 1;
        navigate_left_motor = line_track_left_motor>>LINE_TRACK_FACTOR;
        navigate_right_motor = line_track_right_motor>>LINE_TRACK_FACTOR;
        navigate_motor_flag = 1; //will be set low by turn function
      }

      //watch for non-turn-side branch entire time. don't check across-ness here (x iff last_station==DEER)
      if ( nav_left_reading && last_nav_left_reading && second_last_nav_left_reading==0 ){
        //set station flags only if detection is before turn sense
        if( turn_sensed == 0 ){
```

```
            deer_flag = 1;
            #ifdef DEBUG
              printf("found the DEER\n");
            #endif
          }
          //if branch detected after turn sense, set close flag
          else{ //turn_sensed
            close_left_turn = 1;
          }
        }
        second_last_nav_left_reading = last_nav_left_reading;

        //watch for turn-side non-turn
        if( rabbit_flag == 0 && nav_right_reading && last_nav_right_reading){
          rabbit_flag = 1;
          #ifdef DEBUG
            printf("found the RABBIT\n");
          #endif
          branch_sense_time = mseconds();
        }

        if( turn_sensed ){ //stuff for after turn sensed
          back_nav_left_reading = analog( back_nav_left_sensor ) < back_nav_left_tolerance;
          back_nav_right_reading = analog( back_nav_right_sensor ) < back_nav_right_tolerance;

          //watch for close branches after turn detected
          if( back_nav_left_reading && last_back_nav_left_reading ){
            close_right_turn = 1;
          }

          // watch for across branches of turn(new branch will have already been detected) (x iff last_station==DEER)
          if( nav_left_reading && last_nav_left_reading &&
            nav_right_reading && last_nav_right_reading ){
            deer_duck_flag = 1;
            deer_flag = 1;
            duck_flag = 1;

            #ifdef DEBUG
              printf("set deer_duck_flag\n");
            #endif
          }

          //exit loop for turn after specified time
          if( back_nav_right_reading && last_back_nav_right_reading ){
            navigate_motor_flag = 0;
            break;
          }

          last_back_nav_left_reading = back_nav_left_reading;
          last_back_nav_right_reading = back_nav_right_reading;
        }//end turn_sensed
        last_nav_right_reading = nav_right_reading;
        last_nav_left_reading = nav_left_reading;
      }//end while(1)
    }//end close_right_turn

    //clean up sense memory
    last_nav_left_reading = 0;
    last_nav_right_reading = 0;

    second_last_nav_left_reading = 0;
    second_last_nav_right_reading = 0;

    last_back_nav_left_reading = 0;
    last_back_nav_right_reading = 0;
}//end travel_down_to_opposing_duck

//traverse main line to duck
//called by IR->DUCK and DEER->DUCK
void travel_up_to_opposing_duck(){
```

```
if( close_left_turn ){ //use close turn
  //using close turn maintains it
  while(1){
    nav_right_reading = analog( nav_right_sensor ) < nav_right_tolerance;
    back_nav_left_reading = analog( back_nav_left_sensor ) < back_nav_left_tolerance;

    //use line tracking to stay centered
    navigate_left_motor = line_track_left_motor>>LINE_TRACK_FACTOR;
    navigate_right_motor = line_track_right_motor>>LINE_TRACK_FACTOR;
    navigate_motor_flag = 1;

    //watch for close turn (no chance if station==IR_STATION)
    if ( nav_right_reading && last_nav_right_reading ){
      close_right_turn = 1;
    }

    //sense turns w/ rear sensors
    if( back_nav_left_reading && last_back_nav_left_reading ){
      break;
    }

    last_nav_right_reading = nav_right_reading;
    last_back_nav_left_reading = back_nav_left_reading;
  }// end while(1)
  navigate_motor_flag = 0;
}
else{ //don't use close turn
  //not using a close turn negates it
  while( 1 ){
    nav_left_reading = analog(nav_left_sensor) < nav_left_tolerance;
    nav_right_reading = analog(nav_right_sensor) < nav_right_tolerance;

    //mark when turn first seen
    if( turn_sensed==0 && nav_left_reading && last_nav_left_reading ){
      turn_sensed = 1;
      navigate_left_motor = line_track_left_motor>>LINE_TRACK_FACTOR;
      navigate_right_motor = line_track_right_motor>>LINE_TRACK_FACTOR;
      navigate_motor_flag = 1; //will be set low by turn function
    }

    //watch for new branch entire time (x iff station==IR_STATION)
    if ( nav_right_reading && last_nav_right_reading ){
      //set station flags only if detection is before turn sense
      if( turn_sensed == 0 ){
        irstation_flag = 0;
        #ifdef DEBUG
          printf("lost the IR_STATION\n");
        #endif
      }
      //if branch detected after turn sense, set close flag
      else{ //turn_sensed
        close_right_turn = 1;
      }
    }

    if( turn_sensed ){//stuff for after turn sensed
      back_nav_right_reading = analog( back_nav_right_sensor ) < back_nav_right_tolerance;
      back_nav_left_reading = analog( back_nav_left_sensor ) < back_nav_left_tolerance;

      //watch for close rear branches after turn detected (x iff station==IR_STATION)
      if( back_nav_right_reading && last_nav_right_reading ){
        close_left_turn = 1;
      }

      //exit loop for turn when seen by back nav
      if( back_nav_left_reading && last_back_nav_left_reading ){
        break;
      }
```

```
            last_back_nav_right_reading = back_nav_right_reading;
            last_back_nav_left_reading = back_nav_left_reading;
          }
          last_nav_right_reading = nav_right_reading;
          last_nav_left_reading = nav_left_reading;
        }// end while (1)
    }//end close_turn_left

    //clean up sense memory
    last_nav_left_reading = 0;
    last_nav_right_reading = 0;

    second_last_nav_left_reading = 0;
    second_last_nav_right_reading = 0;

    last_back_nav_left_reading = 0;
    last_back_nav_right_reading = 0;
}//end travel_up_to_opposing_duck

//traverse main line to duck
//called for RABBIT->DUCK
void travel_down_to_adjacent_duck(){

    //set flag for branch that will be missed by front navigation,
    //reset close_left_turn and it will be re-detected if needed
    if( close_left_turn ){
      if( irstation_flag ){
        deer_flag = 1;
        #ifdef DEBUG
          printf("found DEER\n");
        #endif
      }
      else{
        irstation_flag = 1;
        #ifdef DEBUG
          printf("found IR_STATION\n");
        #endif
      }
      close_left_turn = 0;
    }

    while( 1 ){
      nav_left_reading = analog(nav_left_sensor) < nav_left_tolerance;
      nav_right_reading = analog(nav_right_sensor) < nav_right_tolerance;

      // set across flag if left and right overlap (new branch will have already been detected)
      if( nav_left_reading && last_nav_left_reading &&
        nav_right_reading && last_nav_right_reading){
        if( irstation_flag == 0 ){
          irstation_duck_flag = 1;
          irstation_flag = 1;
          duck_flag = 1;
          across_sense_time = mseconds();
          #ifdef DEBUG
            printf("set irstation_duck_flag\n");
          #endif
        }
        else{ //irstation_flag == 1
          if( mseconds() > across_sense_time + BRANCH_SENSE_BLOCK_MSECS){
            deer_duck_flag = 1;
            deer_flag = 1;
            duck_flag = 1;
            #ifdef DEBUG
              printf("set deer_duck_flag\n");
            #endif
          }
        }
      }

      //mark when turn first seen
```

```
    if( turn_sensed==0 && nav_right_reading ){
      turn_sensed = 1;
      navigate_left_motor = line_track_left_motor>>LINE_TRACK_FACTOR;
      navigate_right_motor = line_track_right_motor>>LINE_TRACK_FACTOR;
      navigate_motor_flag = 1; //will be set low by turn function
    }

    //watch for new branches entire time
    if ( nav_left_reading && last_nav_left_reading && second_last_nav_left_reading==0 ){
      //debounce branch sense
      if( mseconds() > branch_sense_time + BRANCH_SENSE_BLOCK_MSECS ){
        branch_sense_time = mseconds();
        //set station flags only if detection is before turn sense
        if( turn_sensed == 0 ){
          if( irstation_flag == 0 ){
            irstation_flag = 1;
            #ifdef DEBUG
              printf("found the IR_STATION\n");
            #endif
          }
          else{ //irstation_flag==1
            deer_flag = 1;
            #ifdef DEBUG
              printf("found the DEER\n");
            #endif
          }
        }
      }

      //if branch detected after turn sense, set close flag
      else{ //turn_sensed
        close_left_turn = 1;
      }
    }
    second_last_nav_left_reading = last_nav_left_reading;

    if( turn_sensed ){//stuff for after turn detected
      back_nav_left_reading = analog( back_nav_left_sensor ) < back_nav_left_tolerance;
      back_nav_right_reading = analog( back_nav_right_sensor ) < back_nav_right_tolerance;

      //watch for close rear branches after turn detected
      if( back_nav_left_reading && last_back_nav_left_reading ){
        close_right_turn = 1;
      }

      //exit loop for turn after specified time
      if( back_nav_right_reading && last_back_nav_right_reading ){
        break;
      }

      last_back_nav_left_reading = back_nav_left_reading;
      last_back_nav_right_reading = back_nav_right_reading;
    }
    last_nav_left_reading = nav_left_reading;
    last_nav_right_reading = nav_right_reading;
  }//end while(1)

  //clean up sense memory
  last_nav_left_reading = 0;
  last_nav_right_reading = 0;

  second_last_nav_left_reading = 0;
  second_last_nav_right_reading = 0;

  last_back_nav_left_reading = 0;
  last_back_nav_right_reading = 0;
}//end travel_to_adjacent_duck

//traverse main line to deer
//called for IR->DEER
```

```
void travel_down_to_adjacent_deer(){
  //set flag for branch that will be missed by front navigation,
  //reset close_right_turn and it will be re-detected if needed
  if( close_right_turn ){
    if( rabbit_flag ){
      duck_flag = 1;
      #ifdef DEBUG
        printf("found DUCK\n");
      #endif
    }
    else{
      rabbit_flag = 1;
      #ifdef DEBUG
        printf("found RABBIT\n");
      #endif
    }
    close_right_turn = 0;
  }

  while( 1 ){
    nav_left_reading = analog(nav_left_sensor) < nav_left_tolerance;
    nav_right_reading = analog(nav_right_sensor) < nav_right_tolerance;

    // set across flag if left and right overlap (new branch will have already been detected)
    if( nav_left_reading && last_nav_left_reading &&
      nav_right_reading && last_nav_right_reading){
      if( rabbit_flag == 0 ){
        deer_rabbit_flag = 1;
        deer_flag = 1;
        rabbit_flag = 1;
        across_sense_time = mseconds();
        #ifdef DEBUG
          printf("set deer_rabbit_flag\n");
        #endif
      }
      else{ //rabbit_flag == 1
        if( mseconds() > across_sense_time + BRANCH_SENSE_BLOCK_MSECS ) {
          deer_duck_flag = 1;
          deer_flag = 1;
          duck_flag = 1;
          #ifdef DEBUG
            printf("set deer_duck_flag\n");
          #endif
        }
      }
    }

    //mark when turn first seen
    if( turn_sensed==0 && nav_left_reading && last_nav_left_reading){
      turn_sensed = 1;
      navigate_left_motor = line_track_left_motor>>LINE_TRACK_FACTOR;
      navigate_right_motor = line_track_right_motor>>LINE_TRACK_FACTOR;
      navigate_motor_flag = 1; //will be set low by turn function
    }

    //watch for new branches entire time
    if ( nav_right_reading && last_nav_right_reading && second_last_nav_right_reading==0){
      //debounce branch sense
      if( mseconds() > branch_sense_time + BRANCH_SENSE_BLOCK_MSECS ){
        branch_sense_time = mseconds();
        //set station flags only if detection is before turn sense
        if( turn_sensed == 0 ){
          if( rabbit_flag == 0 ){
            rabbit_flag = 1;
            #ifdef DEBUG
              printf("found the RABBIT\n");
            #endif
          }
          else{ //rabbit_flag==1
            duck_flag = 1;
```

```
                #ifdef DEBUG
                  printf("found the DUCK\n");
                #endif
              }
            }
          }
          //if branch detected after turn sense, set close flag
          else{ //turn_sensed
            close_right_turn = 1;
          }
        }
        second_last_nav_right_reading = last_nav_right_reading;

        if( turn_sensed ){//tests for after turn sensed
          back_nav_right_reading = analog( back_nav_right_sensor ) < back_nav_right_tolerance;
          back_nav_left_reading = analog( back_nav_left_sensor ) < back_nav_left_tolerance;

          //watch for close rear branches after turn detected
          if( back_nav_right_reading && last_back_nav_right_reading ){
            close_left_turn = 1;
          }

          //exit loop for turn after specified time
          if( back_nav_left_reading && last_back_nav_left_reading ){
            break;
          }

          last_back_nav_right_reading = back_nav_right_reading;
          last_back_nav_left_reading = back_nav_left_reading;
        }
        last_nav_right_reading = nav_right_reading;
        last_nav_left_reading = nav_left_reading;
      }//end while(1)

      //clean up sense memory
      last_nav_left_reading = 0;
      last_nav_right_reading = 0;

      second_last_nav_left_reading = 0;
      second_last_nav_right_reading = 0;

      last_back_nav_left_reading = 0;
      last_back_nav_right_reading = 0;
    }//end travel_down_to_adjacent_deer()

    //traverse main line to deer
    //called for DUCK->DEER and RABBIT->DEER
    void travel_up_to_opposing_deer(){

      if( close_right_turn ){ //use right turn
        //using close turn maintains it
        while(1){
          nav_left_reading = analog( nav_left_sensor ) < nav_left_tolerance;
          back_nav_right_reading = analog( back_nav_right_sensor ) < back_nav_right_tolerance;

          //use line tracking to stay centered
          navigate_left_motor = line_track_left_motor>>LINE_TRACK_FACTOR;
          navigate_right_motor = line_track_right_motor>>LINE_TRACK_FACTOR;
          navigate_motor_flag = 1;

          //watch for close turn (no chance if station==IR_STATION)
          if ( nav_left_reading && last_nav_left_reading ){
            close_left_turn = 1;
          }

          //sense turns w/ rear sensors
          if( back_nav_right_reading && last_back_nav_right_reading ){
            break;
          }
```

```
      last_nav_left_reading = nav_left_reading;
      last_back_nav_right_reading = back_nav_right_reading;
    }// end while(1)
    navigate_motor_flag = 0;
  }
  else{ //don't use close turn
    //not using a close turn negates it
    while( 1 ){
      nav_left_reading = analog(nav_left_sensor) < nav_left_tolerance;
      nav_right_reading = analog(nav_right_sensor) < nav_right_tolerance;

      //mark when turn first seen
      if( turn_sensed==0 && nav_right_reading && last_nav_right_reading){
        turn_sensed = 1;
        navigate_left_motor = line_track_left_motor>>LINE_TRACK_FACTOR;
        navigate_right_motor = line_track_right_motor>>LINE_TRACK_FACTOR;
        navigate_motor_flag = 1; //will be set low by turn function
      }

      //watch for new branch entire time (x iff last_station==RABBIT)
      if ( nav_left_reading && last_nav_left_reading ){
        //set station flags only if detection is before turn sense
        if( turn_sensed == 0 ){
          rabbit_flag = 0;
          #ifdef DEBUG
            printf("lost the RABBIT\n");
          #endif
        }
        //if branch detected after turn sense, set close flag
        else{ //turn_sensed
          close_left_turn = 1;
        }
      }

      if( turn_sensed ){ //stuff for after turn sensed
        back_nav_left_reading = analog( back_nav_left_sensor ) < back_nav_left_tolerance;
        back_nav_right_reading = analog( back_nav_right_sensor ) < back_nav_right_tolerance;

        //watch for close rear branches after turn detected (x iff station==RABBIT)
        if( back_nav_left_reading && last_back_nav_left_reading ){
          close_right_turn = 1;
        }

        //exit loop for turn after specified time
        if( back_nav_right_reading && last_back_nav_right_reading ){
          break;
        }

        last_back_nav_left_reading = back_nav_left_reading;
        last_back_nav_right_reading = back_nav_right_reading;
      }

      last_nav_left_reading = nav_left_reading;
      last_nav_right_reading = nav_right_reading;
    }// end while (1)
  }//end close_turn_left

  //clean up sense memory
  last_nav_left_reading = 0;
  last_nav_right_reading = 0;

  second_last_nav_left_reading = 0;
  second_last_nav_right_reading = 0;

  last_back_nav_left_reading = 0;
  last_back_nav_right_reading = 0;
}//end travel_up_to_opposing_deer()

//traverse main line to deer
//called by DUCK->DEER and RABBIT->DEER
```

```
void travel_down_to_opposing_deer(){

  if( close_left_turn && irstation_flag){ //use close turn
    //close turn flag is maintained
    while( 1 ){
      nav_right_reading = analog( nav_right_sensor ) < nav_right_tolerance;
      back_nav_left_reading = analog( back_nav_left_sensor ) < back_nav_left_tolerance;

      //use line tracking to stay centered
      navigate_left_motor = line_track_left_motor>>LINE_TRACK_FACTOR;
      navigate_right_motor = line_track_right_motor>>LINE_TRACK_FACTOR;
      navigate_motor_flag = 1;

      //watch for a close branch
      if( nav_right_reading && last_nav_right_reading ){
        close_right_turn = 1;
      }

      //take turn immediately after sensing
      if( back_nav_left_reading && last_back_nav_left_reading ){
        break;
      }

      last_nav_right_reading = nav_right_reading;
      last_back_nav_left_reading = back_nav_left_reading;
    }//end while(1)
    navigate_motor_flag = 0;
  }
  else{ //use no close turn

    //make sure that if there was a close turn not taken, the flag is set appropriately
    if( close_left_turn ){
      irstation_flag = 1;
      #ifdef DEBUG
       printf("found the IR_STATION\n");
      #endif
      close_left_turn = 0;
    }

    //look for desired turn after other turn passed
    if( irstation_flag ){
      branch_sense_time = mseconds() - BRANCH_SENSE_BLOCK_MSECS;
    }
    else{
      branch_sense_time = 0L;
    }
    while(1){
      nav_left_reading = analog(nav_left_sensor) < nav_left_tolerance;
      nav_right_reading = analog(nav_right_sensor) < nav_right_tolerance;

      //mark when turn first seen
      if( nav_left_reading &&
        last_nav_left_reading &&
        branch_sense_time > 0L &&
        (mseconds() > branch_sense_time + BRANCH_SENSE_BLOCK_MSECS) ){
        turn_sensed = 1;
        navigate_left_motor = line_track_left_motor>>LINE_TRACK_FACTOR;
        navigate_right_motor = line_track_right_motor>>LINE_TRACK_FACTOR;
        navigate_motor_flag = 1; //will be set low by turn function
      }

      //watch for non-turn-side branch entire time. don't check across-ness here (x iff last_station==DUCK)
      if ( nav_right_reading && last_nav_right_reading && second_last_nav_right_reading==0){
        //set station flags only if detection is before turn sense
        if( turn_sensed == 0 ){
          duck_flag = 1;
          #ifdef DEBUG
           printf("found the DUCK\n");
          #endif
        }
```

```
            //if branch detected after turn sense, set close flag
            else{ //turn_sensed
              close_right_turn = 1;
            }
          }
          second_last_nav_right_reading = last_nav_right_reading;

          //watch for turn-side non-turn
          if( irstation_flag == 0 && nav_left_reading && last_nav_left_reading){
            irstation_flag = 1;
            #ifdef DEBUG
              printf("found the IR_STATION\n");
            #endif
            branch_sense_time = mseconds();
          }

          if( turn_sensed ){ //stuff for after turn sensed
            back_nav_left_reading = analog( back_nav_left_sensor ) < back_nav_left_tolerance;
            back_nav_right_reading = analog( back_nav_right_sensor ) < back_nav_right_tolerance;

            //watch for close branches after turn detected
            if( back_nav_right_reading && last_back_nav_right_reading ){
              close_left_turn = 1;
            }

            // watch for across branches of turn(new branch will have already been detected) (x iff last_station==DUCK)
            //irstation across flags were already set
            if( nav_left_reading && last_nav_left_reading &&
              nav_right_reading && last_nav_right_reading){
              deer_duck_flag = 1;
              duck_flag = 1;
              #ifdef DEBUG
                printf("set deer_duck_flag\n");
              #endif
            }

            //exit loop for turn after sensed by back nav
            if( back_nav_left_reading && last_back_nav_left_reading ){
              break;
            }

            last_back_nav_left_reading = back_nav_left_reading;
            last_back_nav_right_reading = back_nav_right_reading;
          }//end turn_sensed

          last_nav_left_reading = nav_left_reading;
          last_nav_right_reading = nav_right_reading;
        }//end while(1)
      }//end close_right_turn

      //clean up sense memory
      last_nav_left_reading = 0;
      last_nav_right_reading = 0;

      second_last_nav_left_reading = 0;
      second_last_nav_right_reading = 0;

      last_back_nav_left_reading = 0;
      last_back_nav_right_reading = 0;
    }//end travel_down_to_opposing_deer

    //traverse main line to rabbit
    //called by IR->RABBIT and DEER->RABBIT
    void travel_down_to_opposing_rabbit(){
      if( close_right_turn ){ //use close turn
        //using close maintains it
        while(1){
          nav_left_reading = analog( nav_left_sensor ) < nav_left_tolerance;
          back_nav_right_reading = analog( back_nav_right_sensor ) < back_nav_right_tolerance;
```

```
    //use line tracking to stay centered
    navigate_left_motor = line_track_left_motor>>LINE_TRACK_FACTOR;
    navigate_right_motor = line_track_right_motor>>LINE_TRACK_FACTOR;
    navigate_motor_flag = 1;

    //watch for close turn (no chance if station==DEER)
    if ( nav_left_reading && last_nav_left_reading ){
      close_left_turn = 1;
    }

    //sense turns w/ rear sensors
    if( back_nav_right_reading && last_back_nav_right_reading ){
      break;
    }

    last_nav_left_reading = nav_left_reading;
    last_back_nav_right_reading = back_nav_right_reading;
  }// end while(1)
  navigate_motor_flag = 0;
}
else{ //don't use close turn
  //not using a close turn negates it
  while( 1 ){
    nav_left_reading = analog(nav_left_sensor) < nav_left_tolerance;
    nav_right_reading = analog(nav_right_sensor) < nav_right_tolerance;

    //mark when turn first seen
    if( turn_sensed==0 && nav_right_reading && last_nav_right_reading){
      turn_sensed = 1;
      navigate_left_motor = line_track_left_motor>>LINE_TRACK_FACTOR;
      navigate_right_motor = line_track_right_motor>>LINE_TRACK_FACTOR;
      navigate_motor_flag = 1; //will be set low by turn function
    }

    //watch for new branch entire time (x iff station==DEER)
    if ( nav_left_reading && last_nav_left_reading ){
      //set station flags only if detection is before turn sense
      if( turn_sensed == 0 ){
        deer_flag = 1;
        #ifdef DEBUG
          printf("found the DEER\n");
        #endif
      }
      //if branch detected after turn sense, set close flag
      else{ //turn_sensed
        close_left_turn = 1;
      }
    }

    if( turn_sensed ){//stuff for after turn sensed
      back_nav_left_reading = analog( back_nav_left_sensor ) < back_nav_left_tolerance;
      back_nav_right_reading = analog( back_nav_right_sensor ) < back_nav_right_tolerance;

      //watch for close rear branches after turn detected (x iff station==DEER)
      if( back_nav_left_reading && last_back_nav_left_reading ){
        close_right_turn = 1;
      }

      //exit loop for turn after specified time
      if( back_nav_right_reading && last_back_nav_right_reading ){
        break;
      }

      last_back_nav_left_reading = back_nav_left_reading;
      last_back_nav_right_reading = back_nav_right_reading;
    }
    last_nav_left_reading = nav_left_reading;
    last_nav_right_reading = nav_right_reading;
  }// end while (1)
}//end close_turn_left
```

```
  //clean up sense memory
  last_nav_left_reading = 0;
  last_nav_right_reading = 0;

  second_last_nav_left_reading = 0;
  second_last_nav_right_reading = 0;

  last_back_nav_left_reading = 0;
  last_back_nav_right_reading = 0;
}// end travel_down_to_opposing_rabbit

//traverse main line to rabbit
//called by IR->RABBIT and DEER->RABBIT
void travel_up_to_opposing_rabbit(){

  if( close_left_turn && duck_flag==0){ //use close turn
    //close turn is maintained
    while( 1 ){
      nav_right_reading = analog( nav_right_sensor ) < nav_right_tolerance;
      back_nav_left_reading = analog( back_nav_left_sensor ) < back_nav_left_tolerance;

      //use line tracking to stay centered
      navigate_left_motor = line_track_left_motor>>LINE_TRACK_FACTOR;
      navigate_right_motor = line_track_right_motor>>LINE_TRACK_FACTOR;
      navigate_motor_flag = 1;

      //watch for a close branch
      if( nav_right_reading && last_nav_right_reading ){
        close_right_turn = 1;
      }

      //take turn immediately after sensing
      if( back_nav_left_reading && last_back_nav_left_reading ){
        break;
      }

      last_nav_right_reading = nav_right_reading;
      last_back_nav_left_reading = back_nav_left_reading;
    }//end while(1)
    navigate_motor_flag = 0;
  }
  else{ //use no close turn
    //make sure that if there was a close turn not taken, the flag is set appropriately
    if( close_left_turn ){
      duck_flag = 0;
      #ifdef DEBUG
        printf("lost the DUCK\n");
      #endif
      close_left_turn = 0;
    }

    //check if next turn is to be taken
    if( duck_flag==0 ){
      branch_sense_time = mseconds() - BRANCH_SENSE_BLOCK_MSECS;
    }
    else{
      branch_sense_time = 0L;
    }
    while(1){
      nav_left_reading = analog(nav_left_sensor) < nav_left_tolerance;
      nav_right_reading = analog(nav_right_sensor) < nav_right_tolerance;

      //mark when turn first seen
      if( nav_left_reading &&
        last_nav_left_reading &&
        (mseconds() > branch_sense_time + BRANCH_SENSE_BLOCK_MSECS) ){
        turn_sensed = 1;
        navigate_left_motor = line_track_left_motor>>LINE_TRACK_FACTOR;
        navigate_right_motor = line_track_right_motor>>LINE_TRACK_FACTOR;
```

```
        navigate_motor_flag = 1; //will be set low by turn function
      }

      //watch for non-turn-side branch entire time. don't check across-ness here (x iff last_station==IR_STATION)
      if ( nav_right_reading && last_nav_right_reading && second_last_nav_right_reading==0 ){
        //set station flags only if detection is before turn sense
        if( turn_sensed == 0 ){
          irstation_flag = 0;
          #ifdef DEBUG
            printf("lost the IR_STATION\n");
          #endif
        }
        //if branch detected after turn sense, set close flag
        else{ //turn_sensed
          close_right_turn = 1;
        }
      }
      second_last_nav_right_reading = last_nav_right_reading;

      //watch for turn-side non-turn
      if( duck_flag == 1 && nav_left_reading && last_nav_left_reading){
        duck_flag = 0;
        #ifdef DEBUG
          printf("lost the DUCK\n");
        #endif
        branch_sense_time = mseconds();
      }

      if( turn_sensed ){ //stuff for after turn sensed
        back_nav_right_reading = analog( back_nav_right_sensor ) < back_nav_right_tolerance;
        back_nav_left_reading = analog( back_nav_left_sensor ) < back_nav_left_tolerance;

        //watch for close branches after turn detected
        if( back_nav_right_reading && last_back_nav_right_reading ){
          close_left_turn = 1;
        }

        // watch for across branches of turn(new branch will have already been detected) (x iff last_station==IR_STATION)
        // --removed
        // don't care anymore about setting irstation_rabbit_flag

        //exit loop for turn after detected by back nav
        if( back_nav_left_reading && last_back_nav_left_reading ){
          break;
        }

        last_back_nav_right_reading = back_nav_right_reading;
        last_back_nav_left_reading = back_nav_left_reading;
      }//end turn_sensed
      last_nav_right_reading = nav_right_reading;
      last_nav_left_reading = nav_left_reading;
    }//end while(1)
  }//end close_right_turn

  //clean up sense memory
  last_nav_left_reading = 0;
  last_nav_right_reading = 0;

  second_last_nav_left_reading = 0;
  second_last_nav_right_reading = 0;

  last_back_nav_left_reading = 0;
  last_back_nav_right_reading = 0;
}// end travel_up_to_opposing_rabbit

//traverse main line to rabbit
//called by DUCK->RABBIT
void travel_up_to_adjacent_rabbit(){

  //set flag for branch that will be missed by front navigation,
```

```
//reset close_left_turn and it will be re-detected if needed
if( close_right_turn ){
  if( deer_flag ){
    deer_flag = 0;
    #ifdef DEBUG
     printf("lost DEER\n");
    #endif
  }
  else{
    irstation_flag = 0;
    #ifdef DEBUG
     printf("lost IR_STATION\n");
    #endif
  }
  close_right_turn = 0;
}

while( 1 ){
  nav_left_reading = analog(nav_left_sensor) < nav_left_tolerance;
  nav_right_reading = analog(nav_right_sensor) < nav_right_tolerance;

  //mark when turn first seen
  if( turn_sensed==0 && nav_left_reading && last_nav_left_reading){
    turn_sensed = 1;
    navigate_left_motor = line_track_left_motor>>LINE_TRACK_FACTOR;
    navigate_right_motor = line_track_right_motor>>LINE_TRACK_FACTOR;
    navigate_motor_flag = 1; //will be set low by turn function
  }

  //watch for new branches entire time
  if ( nav_right_reading && last_nav_right_reading && second_last_nav_right_reading==0 ){
    //debounce branch sense
    if( mseconds() > branch_sense_time + BRANCH_SENSE_BLOCK_MSECS ){
      branch_sense_time = mseconds();
      //set station flags only if detection is before turn sense
      if( turn_sensed == 0 ){
        if( deer_flag == 1 ){
          deer_flag = 0;
          #ifdef DEBUG
           printf("lost the DEER\n");
          #endif
        }
        else{ //deer_flag==0
          irstation_flag = 0;
          #ifdef DEBUG
           printf("lost the IR_STATION\n");
          #endif
        }
      }
    }

    //if branch detected after turn sense, set close flag
    else{ //turn_sensed
      close_right_turn = 1;
    }
  }
  second_last_nav_right_reading = last_nav_right_reading;

  /*      // set across flag if left and right overlap (new branch will have already been detected)
  //no need to clock since the branch flag is clocked above -- FALSE
  //this is somewhat redundant here - really we just want down passes to catch double crossings
  if( nav_left_reading && last_nav_left_reading &&
    nav_right_reading && last_nav_right_reading ){
    if( irstation_flag == 0 ){
      irstation_rabbit_flag = 1;
      #ifdef DEBUG
       printf("set irstation_rabbit_flag\n");
      #endif
    }
    else{ //irstation_flag == 1
```

```
            deer_rabbit_flag = 1;
            #ifdef DEBUG
              printf("set deer_rabbit_flag\n");
            #endif
          }
        }
*/
        if( turn_sensed ){//stuff for after turn sensed
          back_nav_right_reading = analog( back_nav_right_sensor ) < back_nav_right_tolerance;
          back_nav_left_reading = analog( back_nav_left_sensor ) < back_nav_left_tolerance;

          //watch for close rear branches after turn detected
          if( back_nav_right_reading && last_back_nav_right_reading ){
            close_left_turn = 1;
          }

          //exit loop for turn after specified time
          if( back_nav_left_reading && last_back_nav_left_reading ){
            break;
          }

          last_back_nav_right_reading = back_nav_right_reading;
          last_back_nav_left_reading = back_nav_left_reading;
        }
        last_nav_right_reading = nav_right_reading;
        last_nav_left_reading = nav_left_reading;
      }//end while(1)

      //clean up sense memory
      last_nav_left_reading = 0;
      last_nav_right_reading = 0;

      second_last_nav_left_reading = 0;
      second_last_nav_right_reading = 0;

      last_back_nav_left_reading = 0;
      last_back_nav_right_reading = 0;
    }// end travel_up_to_adjacent_rabbit
```

5.6 SONAR MODULE CODE

```
/* sonar_module.ic

Description: sonar_stop( int distance) sets sonar_stop_flag to true
        after the passed value of units is reached (units are 1.04mm)

        Hardware Used:
          Digital Out 0
          Digital In 7

        Assumes the existence of globals:
        sonar_stop_flag (set to 0 before call to sonar_stop() )

sonar_stop() signals when the robot is a set distance [or closer] away from some obstacle
*/

void sonar_stop( int distance )
{
  int s;
  int prev = 32000;

  sonar_stop_flag = 0;

  while (1){

    s = sonar();

    if( s < distance && prev < distance){
      sonar_stop_flag = 1;
    }
```

```
    if( s > 440 && prev > 440){
      red_green = 1;
    }
    if( s < 440 && prev < 440){
      red_green = 0;
    }

    prev = s;

    msleep(25L);  //50L good, 25L really fast!
  }
}
```